Concurrent and Communicating Systems
A Java Primer on: Exceptions, Threads, Synchronization and I/O

Thomas Heide Clausen[1], Philippe Chassignet[2]

May 4, 2012

[1]thomas@thomasclausen.org
[2]Philippe.Chassignet@polytechnique.edu

Preface

This present primer on concurrent and communicating programming in Java is intended as an aid for students following INF441 option "NET" and INF557. This primer does not pretend to be exhaustive, nor does it attempt to explain all the philosophical principles behind each construct – rather, the objective is to present necessary programming concepts. This primer, also, does not attempt to replicate the JAVA API documentation, but rather present – by way of an abundance of examples – how the API can be conveniently used, as well as how to avoid certain common pitfalls when writing concurrent and communicating programs in Java.

The authors would like to thank Julien Cervelle for his constructive remarks and contributions to the development of this document – Julien Cervelle and Thomas Nowak were instrumental in developing the chapter on "Channels", as well as the Channel-related sections in the chapters on File Input/Output – and contributed much of the initial content in those.

The authors also want to thank Axel Colin de Verdiere (X08) for his review of several versions of the document.

Chapter 1

A Primer on: Exceptions and Exception Handling in Java

An *exception* is the occurrence of a *special condition* during execution of a program, which disrupts the intended program logic.

Consider the piece of Java code in figure 1.1.

```
1  class DivByZero{
2          public static int divide(int a, int b){
3                  return a/b;
4          }
5          public static void main(String[] args){
6                  int i, j, k, l;
7                  i = divide(4, 2);
8                  j = divide(3, 4);
9                  k = divide(-1, 1);
10                 l = divide(100, 0);
11                 System.out.println("i: " + i +
12                                    "j: " + j +
13                                    "k: " + k +
14                                    "l: " + l);
15         }
16 }
```

Figure 1.1: A simple program, illustrating the need for exception handling.

The method `public static int divide(int a, int b)` is designed to return the integer part of a/b. Thus, the calls from lines 7 and 9 succeed and i=2 and k=-1, respectively.

From line 8, dividing 3 by 4 would result in the fractional number 0.75. But 3/4 is evaluated as 0 (*i.e.,* the integer part of 0.75) and, as `divide(...)` is declared to return

only an integer, there is no way of signaling to the calling code that this return value is not actually accurate. Worse, in line 10, division-by-zero is attempted.....

The function `divide(...)` could attempt to check if the divisor was zero, or if the division would not result in an integer value, but that raises the question of how this "special condition" is to be signaled to the caller? The typical way for a method to signal something to its caller is by way of the return value – which value the method `divide(...)` should return, so as to signal this "special condition" to the caller?

It turns out that the return value space of the method `divide(...)` leaves no unused values that can be used for signaling these special conditions to the caller, without the possibility of that value being mistaken for a valid return value – every integer would be the valid result of a division (the trivial example is the division of the integer by 1). Thus, that option can be immediately discarded, and an alternative must be sought.

In the following sections, the program given in figure 1.1 will be developed so as to include proper exception handling, thereby illustrating the key features and constructs for exception handling in Java.

1.1 Introduction to Exceptions

The concept of *exceptions* is a mechanism whereby a program calling a method can formally and unambiguously distinguish "valid return values" from "signals of occurrence of a special condition". Colloquially speaking, a method would use `return` to indicate that correct processing had completed – and would *"throw an exception"*[1] to signal that a special condition had occurred during execution of the method. This allows the caller to be able to either continue processing (in case of `return`) or to specify proper recovery mechanisms (in case that an exception has been thrown). Figure 1.2 illustrates these possible program flows: either, the method will return correctly (`return`), in which case the instructions specified as part of the normal program flow are executed, or the method throws an exception, in which case the normal program flow is interrupted and the instructions specified as part of the exception handling code are executed.

Obviously, when writing a program, the programmer must write both the code for the normal program flow, as well as the exception handling code - which entails that the programmer must know if a method will be throwing exceptions (and, which exceptions are thrown). Thus, APIs and Class Libraries generally specify for each method, in addition to method arguments and return values, the set of exceptions which may possibly be thrown, as well as their significance.

Conversely, when writing a method to be called, the programmer must identify which "special conditions" can occur, as well as writing the program such that corresponding

[1] The phrase *"raise an exception"* is sometimes used in literature for the same concept, and means exactly the same thing as *"throw an exception"*.

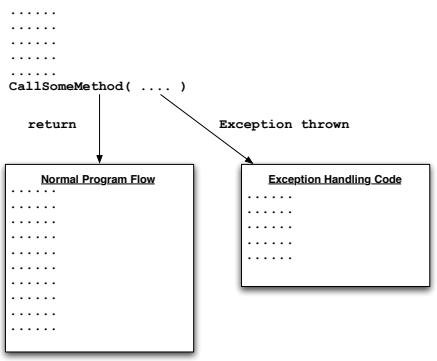


Figure 1.2: Program flow when calling method throws exceptions

exceptions are thrown.

In summary, exceptions entail that:

- A method must be able to indicate that processing succeeded (return) or that a special condition occurred (throw exception).
- The caller of a method must be able to distinguish between that processing succeeded (return was invoked) or that a special condition occurred (exception was thrown).
- In case an exception was thrown, the caller must be able to identify for which reason – and must be able to specify appropriate corrective action in the exception handling code.

1.2 Defining Exceptions in Java

Exceptions in Java are objects. They are distinguishable as such by being instances of the `Exception` class or of classes inherited from the `Exception` class. It is typical that, for each "kind" of exception, a class extending `Exception` is specified. This so as to facilitate that different "special conditions" can be treated differently. For example, if a method both writes to a file on a disk and sends a packet over a network interface, different recovery may be required depending on if it is the exception occurs when the opening the file for writing or when transmitting a packet over the network interface. Declaration of exceptions, permitting signaling the two "special conditions" identified for the program in figure 1.1 ("division by zero", and, "result not an integer") is illustrated in figure 1.3.

It is worth emphasizing that a constructor is declared for each exception, taking a `String` as argument. This enables embedding further details as to program execution at the time when the exception is generated. `super(s)` indicates that the constructor in the

```
1  class DivisorIsZeroException extends Exception{
2
3          DivisorIsZeroException(String s){
4                  super(s);
5          }
6  }
7
8  class ResultNotIntegerException extends Exception{
9
10         ResultNotIntegerException(String s){
11                 super(s);
12         }
13 }
```

Figure 1.3: Declaring Exceptions consists of extending the class Exception

superclass (Exception) is called from the constructor of each of the two exceptions in figure 1.3.

It is also worth emphasizing that this way of declaring exceptions is – almost – mechanical. It is rare to need to do anything more complicated than the above when declaring exceptions.

1.3 Throwing Exceptions in Java – `throws - throw`

In Java, a method which may throw an exception has, as part of its method signature, a statement that it can throw exceptions, as well as which exceptions it can throw. This is strictly analogous to how the message signature must declare the type of the return value.

To continue with the example program from figure 1.1, the method `divide(....)` should be declared with a method signature indicating that it can throw the exceptions `DivisorIsZeroException` and `ResultNotIntegerException`, as in figure 1.4.

```
1          public static int divide (int a, int b)
2                          throws DivisorIsZeroException,
3                                  ResultNotIntegerException{ .... }
```

Figure 1.4: Declaring a Method capable of throwing exceptions of two types

It is worth emphasizing that execution of a method, in Java, can result in either **one** declared return value (in this case, one int) or **one** exception (in this case, either of `DivisorIsZeroException` or `ResultNotIntegerException`). While a method declaration may include a list of several exceptions that this method may throw, any given

execution of the method will throw **at most one** from among these.

Now that the method is declared to be able to throw exceptions, two tasks remain:

- Identify the conditions of execution, where an exception should be thrown
- Initiate the throwing of the corresponding exception.

While the former is an intellectual exercise, which depends on what the program is trying to accomplish, the latter is a mechanical operation. Recall that an Exception in Java is an Object, and that an object is created by way of the operator `new`. In Combining these two, figure 1.5 illustrates how to initiate the throwing of an exception in Java.

```
1        throw new DivisorIsZeroException("Details....");
```

Figure 1.5: "Throwing" an Exception in Java

The keyword `throw` is, for exceptions, the equivalent of the `return` keyword: it halts execution of the method, and returns the specified object. Note that while in Java, it is possible to invoke `return;` without specifying a return value, `throw` **MUST ALWAYS** be followed by the exception object to return.

Figure 1.6 illustrates how the method `divide(....)` from the program in figure 1.1 is augmented to "throw" exceptions.

```
1        public static int divide(int a, int b)
2                                        throws DivisorIsZeroException,
3                                               ResultNotIntegerException{
4                if (b == 0){
5                        throw new DivisorIsZeroException("a="+a+"; b="+b);
6                }
7                if (a%b != 0){
8                        throw new ResultNotIntegerException("a="+a+"; b="+b);
9                }
10
11               return a/b;
12       }
```

Figure 1.6: "Throwing" an Exception in Java

1.4 Exception Handling Code in Java – `try` - `catch`

1.4.1 The `try` - `catch` statement

The last component necessary is writing the exception handling code. In Java, the philosophy is to `try` to execute a method and `catch` exceptions, if they are thrown.

This basic construct is illustrated in figure 1.7.

```
1       try{
2               methodThatCanThrowException();
3       }catch(ExceptionClass e){
4               // Exception handling code here
5       }
```

Figure 1.7: "Catching" an Exception in Java

The class specified in the `catch(ExceptionClass e)` must match the class of exception thrown by `methodThatCanThrowException()`. In case `methodThatCanThrowException()` can throw exceptions of more than one type, then each type must be explicitly caught, as illustrated in figure 1.8.

```
1       try{
2               methodThatCanThrowException();
3       }catch(ExceptionypeOne e){
4               // Exception handling code for Type One Exception here
5       }catch(ExceptionTypeTwo e){
6               // Exception handling code for Type Two Exception here
7       }catch(ExceptionTypeThree e){
8               // Exception handling code for Type Three Exception here
9       }
```

Figure 1.8: "Catching" an Exception of various types in Java

The program flow from figure 1.2 can thus be refined as illustrated in figure 1.9: the method `CallSomeMethod(....)` can `return` and the program can continue processing instructions as part of the normal program flow – or, one of three exceptions may be thrown by `CallSomeMethod(....)`, in which case the normal program flow is interrupted and the instructions specified in the corresponding exception handling code are executed.

Continuing development of the example from figure 1.1, figure 1.6 introduced two exceptions: `DivisorIsZeroException` and `ResultNotIntegerException`. Both of these must be "caught" when `divide(....)` is called, and exception handling code must be written; a first attempt is illustrated in figure 1.10.

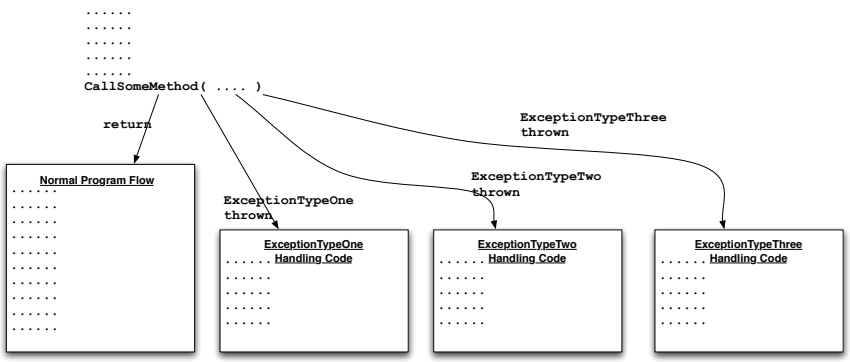


Figure 1.9: Program flow when calling method capable of throwing multiple different exceptions and corresponding exception handling code

```
1       public static void main(String[] args){
2
3               int i, j, k, l;
4               try{
5                       i = divide(4, 2);
6                       j = divide(3, 4);
7                       k = divide(-1, 1);
8                       l = divide(100, 0);
9               }catch(DivisorIsZeroException e){
10                      System.out.println("Divisor was zero: "+e.getMessage());
11              }catch(ResultNotIntegerException e){
12                      System.out.println("Division results in non-integer: "
13                                              + e.getMessage());
14              }
15              System.out.println("i: " + i + " " +
16                                      "j: " + j + " " +
17                                      "k: " + k + " " +
18                                      "l: " + l);
19      }
```

Figure 1.10: Compiler error in "catching" an Exception in Java

What would the execution of the code in figure 1.10 yield? More precisely, which lines of code would be actually executed? It is clear that lines 5 and 7 would "succeed" and that these would not throw any exceptions. It is also clear that executing line 6 should throw a `ResultNotIntegerException` and line 8 should throw a `DivisorIsZeroException`.

All said and done, this program doesn't execute at all – in fact, it doesn't even compile. The Java compiler sees that a valid execution path is lines 1-4, with line 5 throwing an exception when executing `divide(4, 2)` (even before `i` has been assigned a value), continuing with the exception handling in line 9-14, and then through to lines 15-18

```
1       public static void main(String[] args){
2
3               int i=0, j=0, k=0, l=0;
4               try{
5                       i = divide(4, 2);
6                       j = divide(3, 4);
7                       k = divide(-1, 1);
8                       l = divide(100, 0);
9               }catch(DivisorIsZeroException e){
10                      System.out.println("Divisor was zero: "+e.getMessage());
11              }catch(ResultNotIntegerException e){
12                      System.out.println("Division results in non-integer: "
13                                              + e.getMessage());
14              }
15              System.out.println("i: " + i + " " +
16                                      "j: " + j + " " +
17                                      "k: " + k + " " +
18                                      "l: " + l);
19      }
```

Figure 1.11: A poor solution for "catching" an Exception in Java

where the variables `i`, `j`, `k` and `l` are accessed – without these variable having been initialized. The compiler would produce an error akin to the following:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ javac DivTest.java
DivTest.java:80: variable i might not have been initialized
System.out.println("i: " + i + " " +
```

Variables must always be initialized, regardless of execution path, prior to them being accessed. A *poor* solution, which initializes the variables at line 3, is illustrated in figure 1.11. The program can now be compiled and executed.

In actual fact, the lines 1-6 are executed. As the call to `divide` at line 6 results in a `ResultNotIntegerException,` the program flow is *immediately* interrupted, variable `j` is not set and the instructions specified in the handler for `ResultNotIntegerException` in line 11 are executed. Once the exception handler has been executed, the program flow continues at line 14. Specifically the instructions in line 7 and 8 (inside the `try{....}` block) are not executed. Thus, program execution results in the following output, indicating that the exception thrown was the `ResultNotIntegerException` in line 6:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java DivByZero
Division results in non-integer: a=3; b=4
i: 2 j: 0 k: 0 l: 0
```

The rule in Java is, that inside a `try{....}` block, the first exception thrown will cause the program flow to skip immediately to the corresponding exception handler, not executing any further instructions inside that same `try{....}` block. It is therefore

not recommended to follow the example given at line 3 in figure 1.11, because it does not explicitly assign values to the different variables in case an exception is throw. A better solution includes declaring variables without initializing them – as in figure 1.10 – and then let the compiler verify that each case of exception is handled properly in the program.

It is therefore recommended to make each `try{....}` block as "narrow" as necessary, *i.e.,* that contains as few method calls as possible, for which the exceptions may be handled in the same way. Figure 1.12 illustrates this. In this example, it is furthermore illustrated how values can assigned to the corresponding variable in each case of exception; *i.e.,* even in case of exceptions being raised, variables will have well-defined values specified by the exception handing code..

```
1       public static void main(String[] args){
2
3               int i, j, k, l;
4               try{
5                       i = divide(4, 2);
6               }catch(DivisorIsZeroException e){
7                       System.out.println("Divisor was zero: "+e.getMessage());
8                       i = 0;
9               }catch(ResultNotIntegerException e){
10                      System.out.println("Division results in non-integer: "
11                                              + e.getMessage());
12                      i = 2;
13              }
14              try{
15                      j = divide(3, 4);
16              }catch(DivisorIsZeroException e){
17                      System.out.println("Divisor was zero: "+e.getMessage());
18                      j = 0;
19              }catch(ResultNotIntegerException e){
20                      System.out.println("Division results in non-integer: "
21                                              + e.getMessage());
22                      j = 1;
23              }
24              try{
25                      k = divide(-1, 1);
26              }catch(DivisorIsZeroException e){
27                      System.out.println("Divisor was zero: "+e.getMessage());
28                      k = 0;
29              }catch(ResultNotIntegerException e){
30                      System.out.println("Division results in non-integer: "
31                                              + e.getMessage());
32                      k = -1;
33              }
34              try{
35                      l = divide(100, 0);
36              }catch(DivisorIsZeroException e){
37                      System.out.println("Divisor was zero: "+e.getMessage());
38                      l = 0;
39              }catch(ResultNotIntegerException e){
40                      System.out.println("Division results in non-integer: "
41                                              + e.getMessage());
42                      l = 0;
43              }
44              System.out.println("i: " + i + " " +
45                                 "j: " + j + " " +
46                                 "k: " + k + " " +
47                                 "l: " + l);
48      }
```

Figure 1.12: "Catching" an Exception in Java

Execution of the program results in the following output, indicating that both the `ResultNotIntegerException` in line 15 and the `DivisorIsZeroException` in line 35 were thrown – and that program execution otherwise resumed after the exception handling in line 44:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java DivByZero
Division results in non-integer: a=3; b=4
Divisor was zero: a=100; b=0
i: 2 j: 1 k: -1 l: 0
```

1.4.2 Halting Program Execution

It may be the case that it is impossible to recover from an exception, and that the best cause of action is to simply halt the program. For the example in figure 1.12, it may be worthless to recover from a division by zero. To this end, the method `System.exit(int i)` terminates execution of a Java program. The argument given is the *exit code*, by convention a non-zero value indicates abnormal termination of the program (the exact value used is, for the purpose of this primer and for most programs, unimportant). Also note that the compiler does not know that a call to `System.exit(...)` halts the program and does not return – the compiler sees `System.exit(...)` as no different from any other method call in that regard. Thus, the compiler requires that the program contains code forcing that all variables, still, be initialized. Thus, the exception handling from figure 1.12 can be augmented as in figure 1.13. Notice that the assignment of a value to `v` in line 6 actually will never be reached in program execution – but is there simply so that the compiler will not complain that there are uninitialized variables.

```
1   try{
2           v = divide(....);
3   }catch(DivisorIsZeroException e){
4           e.printStackTrace();
5           System.exit(1);
6           v = 0; // not reached
7   }catch(ResultNotIntegerException e){
8           System.out.println("Division results in non-integer: "
9                                   + e.getMessage());
10          v = ...; // a suitable rounded value
11  }
```

Figure 1.13: Halting program execution in Java

As shown in figure 1.13, the `printStackTrace()` method of an object of the `Exception` class can be used to extract further information when an exception is thrown. This leads to an helpful error message – for the complete example of section 1.5, it takes the following form:

```
DivisorIsZeroException: a=100; b=0
    at DivByZero.divide(DivByZero.java:20)
    at DivByZero.main(DivByZero.java:69)
```

Finally, note that the `ResultNotIntegerException` handling code in figure 1.13 displays an error message but permits program execution to continue, with `v` being assigned a suitably rounded value.

1.4.3 Propagating Exceptions in Java

In Java, exceptions thrown when calling a method must be either caught immediately – or the caller must be declared so as to throw that same exception, in which case the exception is said to *"propagate upwards through the call stack"*.

```
1  class Test{
2          public void foobar() throws FooBarException{
3                  // Do something
4                  throw new FooBarException("");
5          }
6
7          public void foo(){
8                  try{
9                          foobar();
10                 }catch (FooBarException e){
11                         // exception handling
12                 }
13         }
14
15         public void bar() throws FooBarException{
16                 foobar();
17                 // some further statements
18         }
19
20         public void doBar(){
21                 try{
22                         bar();
23                 }catch (FooBarException e){
24                         // exception handling
25                 }
26         }
27 }
```

Figure 1.14: Exceptions either caught or propagated

In the example in figure 1.14 the method `foobar()` throws a `FooBarException`. The method `foo()` calls `foobar()` and performs exception handling as previously described, using `try{....} catch(...){....}`.

The method `bar()` also calls `foo()`, but `bar()` does not perform any exception handling.

Method `bar()` is, however, declared to throw a `FooBarException`, so whoever calls `bar()` (for example, `doBar()` must be performing exception handing of `FooBarException`. If a `FooBarException` is thrown when calling `foobar()` from `bar()`, further statements in `bar()` are skipped and this exception is *propagated* to where `bar()` is called, and where it then must be handled – in this case, in the method `doBar()`.

Note that the `main` method may also throw exceptions and thus have the `throws` keyword in its signature. In such case, the exceptions are caught by the *Java Virtual Machine* which causes the immediate termination of the program.

1.5 Complete Divide-By-Zero Example Code

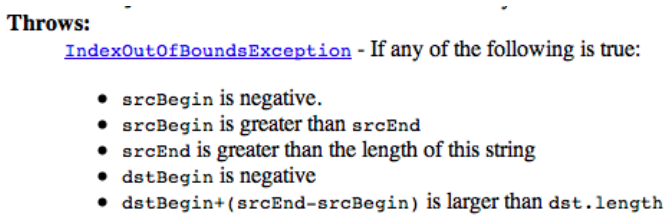
The complete code for the example developed through this chapter is given below, illustrating the main exception handling mechanisms of Java.

```
1  class DivisorIsZeroException extends Exception{
2
3          DivisorIsZeroException(String s){
4                  super(s);
5          }
6  }
7
8  class ResultNotIntegerException extends Exception{
9
10         ResultNotIntegerException(String s){
11                 super(s);
12         }
13 }
14
15 class DivByZero{
16         public static int divide(int a, int b)
17                                         throws DivisorIsZeroException,
18                                                 ResultNotIntegerException{
19                 if (b == 0){
20                         throw new DivisorIsZeroException("a="+a+"; b="+b);
21                 }
22                 if (a%b != 0){
23                         throw new ResultNotIntegerException("a="+a+"; b="+b);
24                 }
25
26                 return a/b;
27         }
28
29         public static void main(String[] args){
30                 int i, j, k, l;
31
32                 try{
33                         i = divide(4, 2);
34                 }catch(DivisorIsZeroException e){
35                         e.printStackTrace();
```

```
36                          System.exit(1);
37                          i = 0; // not reached
38                  }catch(ResultNotIntegerException e){
39                          System.out.println("Division results in non-integer: "
40                                  + e.getMessage());
41                          i = 2;
42                  }
43
44                  try{
45                          j = divide(3, 4);
46                  }catch(DivisorIsZeroException e){
47                          e.printStackTrace();
48                          System.exit(1);
49                          j = 0; // not reached
50                  }catch(ResultNotIntegerException e){
51                          System.out.println("Division results in non-integer: "
52                                  + e.getMessage());
53                          j = 1;
54                  }
55
56                  try{
57                          k = divide(-1, 1);
58                  }catch(DivisorIsZeroException e){
59                          e.printStackTrace();
60                          System.exit(1);
61                          k = 0; // not reached
62                  }catch(ResultNotIntegerException e){
63                          System.out.println("Division results in non-integer: "
64                                  + e.getMessage());
65                          k = -1;
66                  }
67
68                  try{
69                          l = divide(100, 0);
70                  }catch(DivisorIsZeroException e){
71                          e.printStackTrace();
72                          System.exit(1);
73                          l = 0; // not reached
74                  }catch(ResultNotIntegerException e){
75                          System.out.println("Division results in non-integer: "
76                                  + e.getMessage());
77                          l = 0;
78                  }
79
80                  System.out.println("i: " + i + " " +
81                                  "j: " + j + " " +
82                                  "k: " + k + " " +
83                                  "l: " + l);
84          }
85  }
```

1.6 Exceptions and the Java Class Library

Many methods defined by the Java Class Library will throw exceptions, which must be caught and handled. The main way of identifying which methods throw which exceptions is to consult the Java Class Library API documentation, where the exceptions that can be thrown, as well as their interpretation, is specified. As an example, figure 1.15 shows an except of the documentation for the method `getChars(....)` in `java.lang.String`, specifically the exceptions thrown by this method.


Throws:

`IndexOutOfBoundsException` - If any of the following is true:

- `srcBegin` is negative.
- `srcBegin` is greater than `srcEnd`
- `srcEnd` is greater than the length of this string
- `dstBegin` is negative
- `dstBegin+(srcEnd-srcBegin)` is larger than `dst.length`

Figure 1.15: Example of Java Class Library API documentation specifying exceptions thrown (in this case, for `getChars(...)` in `java.lang.String`).

1.7 Special Cases of `Error` and `RuntimeException`

In a broader sense, what was stated above applies to every instance of `Throwable`, the class which represents objects that can be thrown and caught through the Java statements `try{ ...}catch(....){.....}`. The class `Throwable` has only two direct subclasses: `Error` and `Exception`.

An `Error` is thrown to indicate a serious and fatal problem that rises in abnormal conditions. Exemplary subclasses of `Error` are `OutOfMemoryError` and `StackOverflowError`. One should not catch an `Error` since no reasonable handling, other than halting program execution as explained in section 1.4.3, is possible. As an `Error` can be thrown by *any* Java instruction, and as there is no reasonable way to handle such from inside the program, it is better to leave these propagating up to the Java Virtual Machine.

Subclasses of `RuntimeException` are the well known mares of the beginner in Java, such as `ArithmeticException`, `IndexOutOfBoundsException`, `NullPointerException`, ... Such exceptions should not occur in a well thought program with documented and well-tested methods.

It is really a bad idea to try to correct programming errors by catching such exceptions and the only good solution is to prevent them from occurring. Also, undertaking a preliminary test (such as `if (foo != null){ foo.bar(); }` in order to avoid an exception is, generally, not just easier to write, but also much lighter in terms of computation time and memory consumption, than is letting the exception be thrown, then caught and

handled. In this case, too, the best option is to be rigorous in programming and then leave such an exception (should it arrive) propagate up the Java Virtual Machine – which will display a message.

For these reasons, and to simplify writing of programs, it is not required to declare that a methods throws any subclasses of `Error` or `RuntimeException` – they should not happen, and their handling is wired into the final bytecode by the compiler.

An example is given in figure 1.16. In this example, a function has no `throws` clause but still throws an `IllegalArgumentException` – a subclass of `RuntimeException`. To improve its safety, this function validates its actual arguments, instead of allowing errors to occur later. Clearly, if the exception occurs, the problem is from the code that calls this function and not from the function itself.

```
1       public static void doSomeCalculation(int a, int b) {
2               if( a >= b )
3                       throw new IllegalArgumentException("a="+a+"; b="+b);
4               // now, following the calculation that supposes a < b
5               // as indicated in the documentation ...
6               ...
7       }
```

Figure 1.16: A function that throws a `RuntimeException`.

An other example is given in figure 1.17. In this example, an `AssertionError`, a subclass of `Error`, is thrown. In writing this function, we believe that the value of `i` is positive, but it seems difficult to prove. Yet one might have forgotten a particular case, leading to the value of -1 for `i%2` and thus throwing an `IndexOutOfBoundsException`. To express that there is a bad assumption in the code of the function and to facilitate the debugging, it is better to throw an `AssertionError`.

```
1       public static void doSomeCalculation(...) {
2               ...
3               int i = ... // some computed value assumed to be positive
4               if ( i < 0 )
5                       throw new AssertionError("ouch, i >= 0 fails");
6                ... t[i%2] ....
7       }
```

Figure 1.17: A function that throws an `Error`.

1.8 Throw in Catch (Advanced Topic)

The exception handling code inside a `catch`-block is nothing other than regular Java code; all programming constructs, instructions etc. can be used without restrictions. This includes that it is possible for exception handling code to also throw exceptions, for example as illustrated in figure 1.18.

```
1         public static int divide(int a, int b)
2                                         throws DivisorIsZeroException,
3                                                ResultNotIntegerException{
4                 // do calculation, possibly throw exception
5         }
6
7         public static int doCalculation(int a, int b)
8                                         throws UnableToCalculateException{
9                 try{
10                        return divide (a, b);
11                }catch(DivisorIsZeroException e){
12                        // Possibly recover
13                        throw new UnableToCalculateException(e);
14                }catch(ResultNotIntegerException e){
15                        // Possibly recover
16                        throw new UnableToCalculateException(e);
17                }
18
19        }
```

Figure 1.18: Throwing exceptions inside a `catch` block.

As illustrated, `doCalculation(....)` gets a specific exception detailing what went wrong. It may try to recover, or at least bring the system in a coherent state, before it throws a more general exception `UnableToCalculateException` to whoever called `doCalculation(...)`.

1.9 Lazy Exception Handling (Not Recommended)

Exception handling may be tedious, take up many lines of code, and may occasionally appear as an useless exercise. The simple "Division by Zero" example from figure 1.1 started out as just 16 lines of code – with exceptions, section 1.5 had the same example expand to 85 lines of code. Many Java Class Library methods throw exceptions, which must be handled in order for the program to be even able to compile. This may appear tedious, and it may therefore be tempting to do as in figure 1.19.

```
1  class Cheating{
2
3          public static void foo() throws Exception {\\ ... }
4          public static void bar() throws Exception {
5                  foo();
6          }
7
8          public static void main(String[] args) throws Exception{
9                  bar();
10         }
11 }
```

Figure 1.19: Not Recommended: effectively disabling exception handling

Exceptions can propagate through the call stack, as described in section 1.4.3. All exceptions are subclasses of `Exception`, therefore any exception of any type is *also* an exception. Thus, to avoid the perceived tedious task of exception handling, it may appear attractive to declare all methods to simply `throw Exception` – including the method `public static void main(String[] args)`.

Effectively, this entails disabling exceptions and exception handling, propagating all exceptions to `public static void main(String[] args)` – which will propagate the exception to the Java Virtual Machine, reacting to all exceptions by simply terminating the program with an error.

The reason that this is not recommended is, that it renders writing of correct programs difficult, if not impossible. Exceptions are a fact: writing to a disk may fail if there is no more space on the disk, reading a file may fail if the file doesn't exist, opening a network socket may fail if the network is down etc. Such should be handled more gracefully than the program exiting with an error. Exceptions also serve to alert the programmer when doing incorrect operations: reading outside the boundaries of an array (the common mistake of starting at index 1 rather than 0, for example). Simply propagating exceptions to `public static void main(String[] args)` renders it much more complicated to uncover where the exception occurred, and how to debug such errors.

The figure 1.20 gives an example where, by catching every exception, the program-

mer lose the ability to easily find your errors. When running the program all that results is the `"a network error occurred"` message. After spending hours to identify why `doNetwork` raised an exception, finally adding `e.printStackTrace()` in the `catch` clause, makes the programmer realize that the course of the exception is actually not in the method `doNetworkStuff()` in line 10 – but rather than `n` is simply `null` (*i.e.,* line 8 caused the problem) . . . how boring!

```
1  class Network {
2          void doNetworkStuff() throws UnknownHostException, SocketException,
3                                       InterruptedException, InterruptedIOException {
4                  ...
5          }
6
7          public static void main(String[] args) {
8                  Network n = ... // an instruction that fetches a Network object
9                  try {
10                         n.doNetworkStuff();
11                 }
12                 // how boring, I'll catch Exception instead of the four thrown
13                 catch (Exception e) {
14                         System.err.println("a network error occurred");
15                         System.exit(1);
16                 }
17         }
18 }
```

Figure 1.20: Not Recommended: a handler that swallows too many causes

Chapter 2

A Primer on: Java Concurrency, Threads and Synchronization

Java offers two ways of creating threads: by implementing the `Runnable` interface, or by extending the `Thread` class. This primer will illustrate both ways of creating threads, in sections 2.1 and 2.2, respectively. The end result of both is identical – a thread is created. Thus, synchronization mechanisms and life-cycles, as described in further sections, are the same regardless of how they are created.

Note that is always possible for a class to implement the `Runnable` interface and it is, whereas it is not always possible for a class to extend the `Thread` class: as Java doesn't offer multiple inheritance, if the class being used to instantiate a thread must inherit from (`extend`) another class, it cannot also extend the `Thread` class, leaving implementing `Runnable` as the only option. Anyway, implementing the `Runnable` interface is recommended in most cases, for reasons that will be briefly discussed in section 2.3.

2.1 Creating threads by implementing `Runnable`

Interfaces in Java stipulate nothing more than a "contract", to which the programmer must adhere: if a class is declared to `implement theInterface`, then the programmer must implement all the methods specified in the interface declaration of `theInterface`. Objects from a class, implementing a given "interface", allows that object to be treated *as-if* it was of the type of that interface, and in doing so accessing the methods, specified by that interface. Of course, the programmer may freely implement additional methods, not specified in the interface declaration.

The `Runnable` interface in Java is simple, specifying a single method:

```
1  interface Runnable{
2          public void run();
3  }
```

Creating threads by way of the `Runnable` interface is illustrated in figure 2.1.

```
1  class RunnableExample implements Runnable{
2
3          public RunnableExample(){
4                  //      Constructors as usual
5          }
6
7          public void run(){
8                  //      The instructions to execute when the thread is started
9          }
10 }
11
12 class TestRunnable{
13
14         public static void main(String[] args){
15                 //      Create threads t1, t2, in state "new"
16                 Thread t1 = new Thread(new RunnableExample());
17                 Thread t2 = new Thread(new RunnableExample());
18
19                 // Start the threads -- i.e. instruct the scheduler that
20                 // they can be executed -- i.e. they are in state "ready"
21                 t1.start();
22                 t2.start();
23         }
24 }
```

Figure 2.1: Skeleton for creation and starting of threads by way of implementing the `Runnable` interface.

With reference to figure 2.1, note that:

- The `RunnableExample` class, defining the instructions to be executed by the thread, is declared `implements Runnable` and must therefore, at least, contain the method `public void run()` (line 7). That `run()` method is the one executed when the thread is started. One **should not** call directly the `run()` method, it does not start a new thread but just executes the method by the current thread.
- A new thread is created and is given a `run()` method by way of `new Thread( new RunnableExample() )` (lines 16 and 17). This creates a `Thread` object, the constructor of which requires as argument an object of a class which implements the `Runnable` interface (in this case an instance of `RunnableExample`). The thread is not yet running.
- A thread is started by way of invoking the `start()` method on the `Thread` object

(lines 21 and 22). In this example, the instructions `t1.start()` and `t2.start()` instruct the *Java Virtual Machine* that threads `t1` and `t2` are ready for execution and, then, the *JVM scheduler* allocates execution time for each thread on a processor. The instructions executed are those specified in the `run()` method for each thread.

Now, figure 2.2 presents a complete example, based on the skeleton from figure 2.1. The constructor of `RunnableExample2` takes as argument a `String`, and the `run()` method contains a `while(true)` loop, printing that string ad infinitum on the screen.

```
1  class RunnableExample2 implements Runnable{
2
3          String s;
4          public RunnableExample2(String t){
5                  s = t;
6          }
7
8          public void run(){
9                  while (true){
10                         System.out.print(s);
11                 }
12         }
13 }
14
15 class TestRunnable2{
16
17         public static void main(String[] args){
18                 //      Create threads t1, t2, in state "new"
19                 Thread t1 = new Thread(new RunnableExample2("a"));
20                 Thread t2 = new Thread(new RunnableExample2("b"));
21
22                 // Start the threads -- i.e. instruct the scheduler that
23                 // they can be executed -- i.e. they are in state "ready"
24                 t1.start();
25                 t2.start();
26         }
27 }
```

Figure 2.2: Example of two threads, created by way of implementing the `Runnable` interface, respectively printing "a" and "b".

An execution of the program in figure 2.2 program will, depending on the scheduler, produce something like the below:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java RunnableExample
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa .....
```

2.2 Creating threads by extending `Thread`

The second way of creating threads in Java is by way of extending the `Thread` class. The `Thread` class implements the `Runnable` interface, and thus defines a `run()` method by itself – but that method in `Thread` does nothing – contains no instructions. When extending the `Thread` class, the subclass must therefore *override* the `run()` method, in which (just as in section 2.1) the instructions to be executed when the thread starts must be specified. This is illustrated in figure 2.3.

```
1 class ExtendingThread extends Thread{
2         public ExtendingThread(){
3                 //      Constructors as usual
4         }
5
6         public void run(){      //      Override the run() method from Thread
7                 //      The instructions to execute when the thread is started
8         }
9 }
```

Figure 2.3: Creating a thread by extending the `Thread` class

Figure 2.4 presents the exact same functionality as the example in figure 2.2, implemented by way of extending the `Thread` class.

It should be noted in figure 2.4 that the only differences in thread manipulation are that:

- The class defining the thread is declared `extends Thread` (line 1).
- The actual creation of a thread is by way of simply instantiating the declared class (line 19), without a `Runnable` argument.

The rule is: if one does not give the thread an external `run()` method, passed as a `Runnable` argument to the constructor, then the thread uses its own `run()` method. In this case, as the `run()` method of `Thread` does nothing, one must define and use a subclass of `Thread` that overrides the `run()` method.

For completeness, the following way of creating threads – identical to how it is done for threads defined by `implements Runnable` – would still be valid (albeit redundant):

```
1                 Thread t1 = new Thread (new ExtendingThread("a"));
2                 Thread t2 = new Thread (new ExtendingThread("b"));
```

```
1  class ExtendingThread extends Thread{
2
3          String s;
4          public ExtendingThread(String t){
5                  s = t;
6          }
7
8          public void run(){
9                  while (true){
10                         System.out.print(s);
11                 }
12         }
13 }
14
15 class ExtendingThreadExample{
16
17         public static void main(String[] args){
18                 //      Create threads t1, t2, in state "new"
19                 Thread t1 = new ExtendingThread("a");
20                 Thread t2 = new ExtendingThread("b");
21
22                 // Start the threads -- i.e. instruct the scheduler that
23                 // they can be executed, that they now are in runnable state.
24                 t1.start();
25                 t2.start();
26         }
27 }
```

Figure 2.4: Example of two threads, created by way of extending the `Thread` class, respectively printing "a" and "b".

2.3 `extends Thread` or `implements Runnable`?

Given such two ways of creating threads, it is legitimate to reflect upon which should be used when. While there are no hard-and-fast rules, `implements Runnable` is recommended and should be used in most cases since it clearly separates application specific *processing* and thread *execution*:

- Processing is defined by way of an object, of a class that `implements Runnable`, and that therefore defines the `run()` method, defines other methods (called by `run()`) and the required processing state (variables or attributes – *e.g.,* `s`, in the examples given.
- Execution is defined by way of another object, of the `Thread` class, which is used to control the thread lifecycle, *i.e.,* by calling the associated `start()` method (as already seen) as well as `join()`, `interrupt()`, ... methods, which will be seen in the following sections.

When extending `Thread`, the same object plays both of the two roles described above. This may be particularly confusing, since in that particular case `this`, the reference to the actual `Runnable` application specific processing object, and `Thread.currentThread()`, the reference to the thread execution controller object, are interchangeable while they are not actually semantically identical.

The bottom line: use `implements Runnable` unless there is a really really good reason to do otherwise. So, when the objective is only to define the `run()` method, use `implements Runnable`.

2.4 How to accommodate the `run()` method

The `public void run()` signature, declared by the `Runnable` interface can not be modified. If one tries to add parameters, the effect depends on how that is used:

- By way of implementing `Runnable`, one get a compiler error that indicates that the `run()` method is not implemented. It is safe.
- By way of extending `Thread`, one simply overloads the `run()` method by **another** method but, when one will start the thread, the `run()` method – that does nothing – will be called anyway. It is a frequent source of error.

The only way to pass parameters to the `run()` method is through a constructor which stores them in corresponding fields of the object, *e.g.,* `s`, in the above examples.

One can not change the `void` return type. In any case, it is forbidden by the compiler. If the execution should return a result, it can be stored in a structure passed (by reference) to the constructor. Another solution, particularly for a primitive type value, is to leave

the result in a field of the `Runnable` object and to retrieve it later. This assumes that one keeps a reference on the `Runnable` object.

Visibility of the `run()` method must be kept `public`.

Lastly, the `run()` method can not have a `throws` clause. However, it can throw an `Error` or a `RuntimeException`, see section 1.7, that causes the abrupt end of the thread. Whenever possible, the causes of exceptions should be resolved before or during the construction of the `Runnable` object. In the other cases, that can not be prevented, it is always possible to catch and wrap any kind of exception in a `RuntimeException`, as seen in section 1.8.

The following sections will discuss the case of the `InterruptedException` that is related to the control of the thread lifecycle and which requires specific handling.

2.5 Sleep

The class `Thread` contains a static method `Thread.sleep(...)`. Calling this method causes the calling thread, aka the *current thread*, to suspend its execution – to sleep – for the specified number of milliseconds. For example, `Thread.sleep(1000)` will cause the thread calling this method to sleep for *approximately* 1 second. The thread should not resume execution for at least, the time specified in the argument to `Thread.sleep(...)` – scheduling (or imprecision in the operating system clock) may cause the thread to resume execution later.

Note that `sleep` is a static method of class `Thread`. When writing a method, it is impossible to know which thread will actually be executing that method. Thus, a statement such as `t.sleep(...)` for a given `Thread` object `t` would be nonsense. The `Thread` class has an internal mechanism to determine which thread is calling a given method, and will use this to target the appropriate `Thread` object when `Thread.sleep()` is invoked.

The simplest static method of this kind is `Thread.currentThread(...)`, which returns a reference to the `Thread` instance representing the calling thread.

A thread may also be interrupted in its sleep (*e.g.,* by way of the method `interrupt()` being invoked on this thread), and may therefore wake up earlier than specified in the argument to `Thread.sleep`. If a sleeping thread is interrupted, an `InterruptedException` is thrown, which must be handled immediately upon the thread resuming execution (awakening), see section 2.6.

Figure 2.5 illustrates the use of `sleep`, by extending the program from figure 2.2 to require each thread to sleep for a specified amount of milliseconds between each character is printed.

```
1  class SleepingRunnable implements Runnable{
2
3          String s;
4          int interval;
5
6          public SleepingRunnable(String t, int toSleep){
7                  s = t;
8                  interval=toSleep;
9          }
10
11         public void run(){
12                 while (true){
13                         System.out.print(s);
14                         try{
15                                 // any thread passing here will sleep
16                                 Thread.sleep(interval);
17                         }catch(InterruptedException e){
18                                 // Handle interruption
19                         }
20                 }
21         }
22 }
23
24 class SleepingThreadTest{
25
26         public static void main(String[] args){
27                 Thread t1 = new Thread(new SleepingRunnable("a", 100));
28                 Thread t2 = new Thread(new SleepingRunnable("b", 50));
29                 t1.start();
30                 t2.start();
31         }
32 }
```

Figure 2.5: Example of two threads, created by way of implementing the `Runnable` interface, respectively printing "a" and "b" and each thread sleeping (a certain amount of milliseconds) between each loop in `run()`.

Executing the program in figure 2.5 will, depending on the scheduler, produce something like the below (notice, twice as many b's as a's since the thread printing a's sleep twice as much as the thread printing b's):

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java SleepingThreadTest
abbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabb
abbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabb
abbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabb
abbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbabbaba.....
```

2.6 Stopping a Thread

A thread stops when and only when the `run` method returns. There is no explicit primitive to "kill" a thread or to tell a thread to stop – *i.e.,* one cannot do `t1.stop()` to kill the thread `t1`. The reason for this is, that such killing of a thread from "outside of the thread" easily leads to inconsistent shared data. Imagine that a thread is swapping the content of two shared integer variables `a` and `b`, using the following instructions:

```
1        int t = a;
2        a = b;
3        b = t;
```

If this thread is killed between lines 2 and 3, one value is missing and the other is repeated. If a thread has locked resources, how does it release them? We will see how it can be complex in sections 2.8 and following. Thus, the way in which to stop a thread is to instruct it to, as soon as possible, cleanly and safely restore data in a consistent state and then return from its `run()` method.

Note that the Java Class Library API for the class `Thread` contains methods such as `suspend()` and `stop()`, however these **should not** be used as they are considered as **not safe** for the above reasons[1] – these methods are tagged *deprecated*.

Cleanly and safely stopping a thread is no mechanical task, and always requires some care. In this section, a method is illustrated, for signaling to a thread that it should exit its `run()` method, and some considerations are presented as to where care should be taken when implementing such.

One way for a thread to *interrupt* another thread `t1` is simply to invoke the `interrupt()` method on that thread, thus `t1.interrupt()`. One way for the *current thread* to test whether it has been interrupted is to invoke the `Thread.interrupted()` static method. It works by the way of a (boolean) *interrupted status* for each thread. The *interrupted status* is usually set (to `true`) by `interrupt()`. A call to `Thread.interrupted()` returns the value of the *interrupted status* for the *current thread*, **but** also immediately after clears this *interrupted status* (*i.e.,* sets it back to `false`). The way around this annoying feature will be illustrated in next examples.

Figure 2.6 illustrates how `interrupt()` and `Thread.interrupted()` can be used for one thread requesting that another thread terminates its execution. The program in figure 2.6 has two threads: one created when the program is launched (the *main thread*) and one, `t1`, created in line 17 and started in line 18. At some point, the *main thread* determines that `t1` should be terminated – or, rather, that `t1` should be requested to terminate – and thus invokes `t1.interrupt()` in line 20. This causes the *interrupted status* of `t1` being set to `true`. In this example, we consider that shared data are in a consistent state at the end of each turn of a main loop inside the `run()` method and

[1] Actually, `suspend()` has the added bonus of being deadlock-prone too, as the suspended thread may keep locked resources.

thus the loop may be ended after completing any turn, depending on the test of the *interrupted status* at line 5. So `run()` will return and the thread `t1` will terminate. Before that, some terminating instructions, at line 10, should ensure that any shared data is left in a consistent state, that open files are closed, etc.

```
1  class LoopToStop implements Runnable{
2
3          public void run(){
4                  // initialiazing intructions
5                  while ( !Thread.interrupted() ) {
6                          // instructions to execute
7                          // as long as the thread has not been interrupted
8                          // A full cycle is safe
9                  }
10                 // terminating instructions
11         }
12 }
13
14 class TestThreadStopping{
15
16         public static void main(String[] args){
17                 Thread t1 = new Thread(new LoopToStop());
18                 t1.start();
19                 // Do other stuff
20                 t1.interrupt();
21         }
22 }
```

Figure 2.6: A basic mechanism for requesting that a running thread be terminated a loop using `interrupt()` and `Thread.interrupted()`.

Figure 2.7 illustrates now the case of two nested loops. If the *interrupted status* is tested as `true` at line 5, then the inner loop terminates, but as `Thread.interrupted()` also clears the *interrupted status*, subsequent tests of this status would return `false`. Then, the *current thread* must interrupt itself again, at line 8, so that the outer loop also ends, on the next test occurring at line 4.

The scheme of figure 2.7 can also be used in cases where the inner loop is written in another method which is invoked from inside the outer loop. Then note that the systematic invocation of `Thread.currentThread().interrupt()` at line 8 makes senses, as long as one goes out of the inner loop only in the case of an interruption. In a more complicated case, when the inner loop can also end for another cause, resetting the *interrupted status* becomes more tricky.

An alternative way is to explicitly manage a local copy of the *interrupted status*, as shown in figure 2.8. Note that it is a local variable – allocated on the execution stack of the concerned thread – while using an object field would be more ambiguous, for reasons given in section 2.3, and sometimes false. Auxiliary methods that return a value

```
1  class TwoLoopsToStop implements Runnable{
2
3          public void run(){
4                  while ( !Thread.interrupted() ) {
5                          while ( !Thread.interrupted() ) {
6                                  // instructions of the inner loop
7                          }
8                          Thread.currentThread().interrupt();
9                  }
10         }
11 }
```

Figure 2.7: A mechanism for requesting that a running thread be terminated two nested loops using `interrupt()` and `Thread.interrupted()`.

reflecting the *interrupted status* are another safe way to propagate it back.

```
1  class TwoLoopsToStop implements Runnable{
2
3          public void run(){
4                  boolean interrupted = false;
5                  while ( !interrupted ) {
6                          while ( !interrupted ) {
7                                  // instructions of the inner loop
8                                  if (Thread.interrupted())
9                                          interrupted = true;
10                         }
11                 }
12         }
13 }
```

Figure 2.8: A second mechanism for requesting that a running thread be terminated two nested loops using `Thread.interrupted()` and a local variable.

Now, recall from section 2.5, that a thread may also be interrupted while sleeping. If a thread is interrupted while executing `Thread.sleep(...)`, an `InterruptedException` is thrown, which must be handled immediately upon the thread resuming execution (awakening). Catching the `InterruptedException` is the only way to know that the thread was interrupted during its sleep, by distinguishing from a normal return from the `sleep` method when the sleep duration has expired. When an `InterruptedException` is thrown, the *interrupted status* has **not** been set and the minimal role of the `catch` block is to propagate the knowledge of an interruption, in a manner consistent with what is done around, leading to a graceful termination.

Figures 2.9 and 2.10 illustrate how must be written a call to `Thread.sleep()` in both cases, respectively of figures 2.7 and 2.8.

```
1                   try{
2                           Thread.sleep(...);
3                   } catch (InterruptedException e) {
4                           Thread.currentThread().interrupt();
5                   }
```

Figure 2.9: A basic handling of `InterruptedException`, ensuring a graceful termination in the context of figure 2.7.

```
1                   try{
2                           Thread.sleep(...);
3                   } catch (InterruptedException e) {
4                           interrupted = true;
5                   }
```

Figure 2.10: A basic handling of `InterruptedException`, ensuring a graceful termination in the context of figure 2.8.

A point should be made to recall the necessity of exception handling being done properly, by catching *as specific exceptions as possible*. Imagine if the exception handling in figures 2.9 or 2.10 had been done as follows:

```
1                   try{
2                           Thread.sleep(some_expression);
3                   } catch (Exception e) {
4                           // exception handling
5                   }
```

When entering the exception handling code in line 4, it would be much more difficult to determine if the exception was due to an `interrupt()`, and thus a request for graceful termination, or if it was due to some other exception being thrown (such as would be the case in this example, where the evaluation of the expression in line 2 would cause a `DivideByZeroException` to be thrown). It would also render the exception handling code more complicated and less modular.

The bottom line: always catch as specific exceptions as possible.

2.7 Join

In a program with multiple threads, each thread executes independently, except for when explicit synchronization of the execution of two threads is undertaken. The simplest such synchronization is when two threads *join* to become one. More precisely, in a program with two threads, `t1` and `t2`, when thread `t1` invokes the method `t2.join()`, `t1` will wait for `t2` to complete its execution – to terminate – before proceeding. This is illustrated in figure 2.11.

```
1  class SleepingThread extends Thread{
2
3          String s;
4          int interval;
5          Thread joinWithThis;
6
7          public SleepingThread(String t, int toSleep){
8                  this(t, toSleep, null);
9          }
10
11         public SleepingThread(String t, int toSleep, Thread threadToJoin){
12                 s = t;
13                 interval=toSleep;
14                 joinWithThis = threadToJoin;
15         }
16
17         public void run(){
18                 for (int i=0; i<10; i++){
19                         System.out.print(s);
20                         try{
21                                 Thread.sleep(interval);
22                         }catch(InterruptedException e){
23                                 // Handle interruption
24                         }
25                 }
26                 if (joinWithThis != null){
27                         try{
28                                 joinWithThis.join();
29                         }catch(InterruptedException e){
30                                 // Handle interruption
31                         }
32                         System.out.println();
33                         System.out.println("All done");
34                 }
35         }
36 }
37
38 class JoinThreadTest{
39
40         public static void main(String[] args){
41                 Thread t1 = new SleepingThread("a", 100);
42                 Thread t2 = new SleepingThread("b", 50, t1);
43                 t1.start();
44                 t2.start();
45         }
46 }
```

Figure 2.11: Joining of two threads: `t1` and `t2`

Two threads, `t1` and `t2` are created, the latter is given a reference to `t1` in the constructor – with that reference being stored in the local variable `joinWithThis`. Each thread is to print 10 characters to the screen. `t1` prints characters with an interval of 100ms, `t2` with an interval of 50ms. Thus, `t2` will finish first, upon which it will seek to join with the other thread, by way of invoking `joinWithThis.join()`. As a thread, waiting to join another thread, can be interrupted, an `InterruptedException` may thrown, which must be handled.

Executing the program in figure 2.11 will, depending on the scheduler, produce something like the below (notice, the 10 b's are printed at a higher frequency than the 10 a's). When all b's are printed, the thread printing b's will seek to join the other thread – and, upon successful joining (*i.e.,* the other thread terminating), outputting `All done`:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java JoinThreadTest
abbabbabbabbabbaaaaa
All done
```

2.8 Monitors

The synchronization primitive in Java is a variation of a *monitor*. In Java, each object is associated with one MUTEX and one Condition Variable, and methods which require the calling thread to acquire the MUTEX are specified by way of the keyword `synchronized`, as illustrated in figure 2.12.

```
1  class ExampleOfMonitor{
2          public synchronized void foo(){ .... }
3          public synchronized void bar(){ .... }
4          public synchronized void foobar(){ .... }
5  }
```

Figure 2.12: An example of a Monitor in Java

An object of the class `ExampleOfMonitor` from figure 2.12 will allow only one thread to execute any of its methods at any given time. Thus, if a thread, `t1`, wishes to execute the method `foo()`, it is only permitted to do so in case no other threads are at the same time executing either of `foo()`, `bar()` or `foobar()`.

Thus, if an object encapsulates some shared data as `private` attributes, and provides access to these data only by way of synchronized accessor methods, then mutual exclusion is provided. Figure 2.13 shows an extremely simple example of a Java monitor encapsulating a piece of shared data – the `private int mySharedValue` – and provides access to this data only by synchronized accessor methods

```
1  class ExampleOfMonitor{
2          private int mySharedValue;
3          public synchronized void set (int i){
4                  mySharedValue=i;
5          }
6          public synchronized int get(){
7                  return mySharedValue;
8          }
9          public synchronized void reset(){
10                 mySharedValue=0;
11         }
12 }
```

Figure 2.13: An example of a Monitor in Java, protecting a piece of shared data and providing synchronized access only by way of accessor methods.

It is possible that some methods are declared `synchronized` while others are not, as illustrated in figure 2.14. In such a case, only a single thread is allowed to execute any of the methods declared as `synchronized` (since these threads will need to acquire the MUTEX) at any time. Methods not declared `synchronized` do not attempt to acquire

the MUTEX, and thus execution of these is not restricted by the number of threads executing these or other methods.

```
1  class ExampleOfMonitor{
2          private int mySharedValue;
3          public synchronized void set (int i){
4                  mySharedValue=i;
5          }
6          public synchronized int get(){
7                  return mySharedValue;
8          }
9          public string sayHello(){
10                 return "Hello";
11         }
12 }
```

Figure 2.14: An example of a partial Monitor in Java – not all methods are declared `synchronized`

It is generally recommended, when writing multi-threaded programs, that classes be declared so that:

- All attributes (variables) are declared `private`.
- All methods are declared `synchronized`.
- Static methods not be used (other than `public static void main (....)`)..

There may be exceptional cases where deviating from the two latter recommendations is reasonable – however such only after carefully having considered the implications (will concurrent access be able to have any impact on proper functioning of the program?).

2.9 Monitor and Condition Variables

A thread, executing a `synchronized` method in an object has, effectively, acquired the MUTEX on that object – *i.e.,* no other thread can execute methods in that object. This ensures MUTual EXclusive access to methods in that object and (provided that all attributes are declared `private`) thereby also to the data encapsulated by the object. This is, generally, a good thing – it avoids issues such as data being garbled due to concurrent access – but, may also pose its own set of problems with respect to the program logic.

Consider the situation where an object encapsulates a buffer with a limited number of places, as illustrated in figure 2.15. The buffer has two operations `get()`, which returns the first Object in the buffer, and `put(Object o)`, which adds the object, given as argument, to the end of the buffer. When `put(Object o)` returns, it is expected that

the Object given as argument has been added to the buffer – and when `get()` returns, it is expected that an Object is, indeed, returned. This entails that:

```
1
2  class BoundedBuffer{
3          // Buffer with capacity of 5 objects
4          public synchronized void add(Object o){
5                  // .....
6          }
7
8          public synchronized Object get(){
9                  // .....
10         }
11
12 }
```

Figure 2.15: Bounded Buffer for Concurrent Access - what is the proper behavior if `add(Object o)` is invoked when there are already 5 elements in buffer?

- If the buffer is empty, `get()` cannot return, and must be blocked until some other thread has invoked `put()`.
- If the buffer is full, `put(Object o)` cannot return, and must be blocked until some other thread has invoked `get()`.

However, with each of the methods `get()` and `put(Object o)` being declared `synchronized`, once a thread has acquired the MUTEX and started executing a method in the object, all other threads trying to execute synchronized methods in the object are blocked, waiting to be able to acquire the MUTEX. The concept of *condition variables* exist in order to facilitate resolving this situation, allowing a thread to:

1. temporarily release the MUTEX, so as to enable another thread to acquire the MUTEX and execute a `synchronized` method in the same object
   (*e.g.,* when the buffer is full, `put()` permits a `get()` to be executed, liberating space in the buffer);
2. suspend (block) the thread, so as to enable the scheduler to not consider the suspended thread from being executed while the thread is *waiting* for a condition to change
   (*e.g.,* that `get()` has been executed and that there now may be space liberated in the buffer for a waiting `put(Object o)` to proceed);
3. wake up and attempt to re-acquire the MUTEX when *notified* that a condition may have changed.

Colloquially speaking, a condition variable serves two logical functions: (i) permits releasing the MUTEX, even while executing inside a synchronized method, and (ii) suspending a running thread until a desired condition is signaled.

In Java, the following instructions operate on condition variables:

`wait()`
: The current thread (invoking `wait()`) releases the MUTEX and suspends itself, waiting for another thread to invoke `notify()` or `notifyAll()`

`notify()`
: Signals to, and awakens, one of the threads (if any), waiting on the condition variable (*i.e.,* a thread having previously invoked `wait()` while executing a method in the same object).

`notifyAll()`
: Signals to, and awakens, all of the threads (if any), waiting on the condition variable (*i.e.,* a thread having previously invoked `wait()` while executing a method in the same object).

There are a couple of points to be made regarding the above. First, each Object in Java has exactly one MUTEX and exactly one condition variable associated. As one of the key points of condition variables is to release the MUTEX of the object, `wait()`, `notify()` and `notifyAll()` all require that they be executed inside a synchronized method[2]. Second, if no threads are waiting on a given condition variable, then `notify()` and `notifyAll()` have no effect – as no threads are suspended, waiting for the condition to change. Third, when a thread is awakened, it will still have to await that the MUTEX is released – the thread invoking `notify()` or `notifyAll()` retains the mutex until it exits the `synchronized` method. Indeed, the awakened thread will contend with all other threads seeking to acquire this MUTEX. Fourth, the distinction between `notify()` and `notifyAll()` is that while the former wakes one of the waiting threads (if there are any), the latter waits **all** the waiting threads. As there is only one condition variable in each Object in Java, it may be that a different threads waits for different conditions (*e.g.,* buffer-space for one element liberated, buffer-space for multiple elements liberated) – and the only safe reason is to awaken them all and have them test if the condition they are waiting for is satisfied or not.

This leads to a fifth observation: a thread being awakened (*i.e.,* `wait()` returns) cannot assume that the condition it was waiting for is satisfied. It may be that another thread was awakened as well and scheduled first (*e.g.,* buffer-space was liberated, but another thread consumed it already), or it may be that the condition change signaled was different from that which the thread was waiting for.

Thus, `wait()` should systematically be used thus:

```
1
2         while (!cond){   // cond is the condition to be satisfied
3                 wait();
4         }                // when awakened, loop and check if condition is
5                          // satisfied, otherwise wait again.
```

[2]Technically: "inside a synchronized block", of which a synchronized method is an example

To illustrate the usage of Condition Variables in Java, two examples will be provided in the following sections: section 2.9.1 will illustrate a relatively simple case of wanting two threads to alternate their access to a shared resource, while section 2.9.2 illustrates a generalization of the same problem, by way of the now classic *Producer-Consumer* problem, key both to illustrating the use of Monitors, as well as to many many real-world examples of concurrent systems.

2.9.1 Example: Alternating Writers

As seen in figure 2.2, two uncoordinated threads printing, respectively, a's and b's generate a somewhat unpredictable interleaving of a's and b's. This is dependent on how the two threads are scheduled by the system. If the desired result is an alternating sequence of a's and b's, explicit coordination between the two threads is required. To this end, an `AlternatingOutput` class is specified, with a `public synchronized print(char c)` method. A monitor, only a single thread at a time can execute inside the methods of an object of `AlternatingOutput`.

When a thread acquires the MUTEX, and thus can execute `print(...)`, the first thing it does is to check if the last printed character corresponds to the character that is to be printed. If yes, then the thread must (i) release the MUTEX and (ii) wait until the other thread has printed its character. This is accomplished by way of the `while (c == last){ wait() }` loop.

`wait()` causes the thread to release the MUTEX and patiently await that another thread signals that "the condition you are waiting on may have changed" – by way of `notify()` in line 30. This `notify()` is executed only after some other thread has printed its character and changed last. Upon receiving a `notify()` the thread re-awakens, and contends to re-acquire the MUTEX before proceeding.

A `while (....)` loop is used, so as to ensure that when the waiting thread awakens, it re-verifies if it is indeed its turn – it may have been awoken for some other reason (interrupt, ...). If not, the thread will `wait()` again – otherwise, it will proceed to execute lines 28-31, *i.e.,* printing its character, changing `last` to record the character just printed, and signaling to any other waiting threads that "the condition you are waiting on may have changed".

Executing the program in figure 2.16 will, independent of the scheduler, produce something like the below – alternating a's and b's in the output ensured by `AlternatingOutput`:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java AlternatingOutputExample
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababa ....
```

```
1  class PrintingThread extends Thread{
2
3          char s;
4          AlternatingOutput out;
5
6          public PrintingThread(char t, AlternatingOutput o){
7                  s = t;
8                  out = o;
9          }
10
11         public void run(){
12                 while (true){
13                         out.print(s);
14                 }
15         }
16 }
17
18 class AlternatingOutput{
19         private char last;
20
21         public synchronized void print(char c){
22                 while (c == last){
23                         try{
24                                 wait();
25                         }catch(InterruptedException e){
26                         }
27                 }
28                 System.out.print(c);
29                 last = c;
30                 notify();
31         }
32 }
33
34 class AlternatingOutputExample{
35
36         public static void main(String[] args){
37                 AlternatingOutput out = new AlternatingOutput();
38                 Thread t1 = new PrintingThread('a', out);
39                 Thread t2 = new PrintingThread('b', out);
40
41                 // Start the threads -- i.e. instruct the scheduler that
42                 // they can be executed, that they now are in runnable state.
43                 t1.start();
44                 t2.start();
45         }
46 }
```

Figure 2.16: An example of a monitor, used to enforce that the two threads `t1` and `t2` alternate in writing a's and b's.

2.9.2 Example: Producer-Consumer

The Producer-Consumer problem is a classic problem in concurrency: a bounded buffer, *i.e.,* a buffer with a fixed and inelastic capacity, provides two methods: `read()` and `write(Object o)` – and two threads seek to access this buffer concurrently.

One thread is the *Producer*, *i.e.,* it writes data to the buffer. If there is free space in the buffer the write succeeds and `write(Object o)` returns, otherwise the Producer-thread is blocked until space becomes available.

The other thread is the *Consumer* thread, which reads data from the buffer. If there is data available in the buffer, the read operation succeeds and `read()` returns, otherwise the Consumer-thread is blocked until data becomes available.

The producer is relatively simple, as illustrated in figure 2.17: it `implements Runnable` in order to enable a thread being created – and the `run()` method simply writes the letters of the alphabet into the queue, one at a time. The constructor takes the shared `Queue` as argument, as well as a parameter specifying the interval between two successive `write(...)`.

```
1  class Producer implements Runnable{
2
3          private Queue theQueue;
4          private final String alphabet = "abcdefghijklmnopqrstuvwxyz";
5          private int interval;
6
7          public Producer(Queue q, int inter){
8                  theQueue = q;
9                  interval = inter;
10         }
11
12         public void run(){
13                 try{
14                         int index = 0;
15                         while(index < alphabet.length()){
16                                 theQueue.write(alphabet.charAt(index));
17                                 index = index + 1;
18                                 Thread.sleep(interval);
19             }
20         }catch(InterruptedException e){
21                 // Exception Handling
22         }
23     }
24 }
```

Figure 2.17: The Producer-class for the Producer-Consumer example.

The consumer is similarly simple, as illustrated in figure 2.18: it also `implements Runnable` in order to enable a thread being created – and the `run()` method simply

reads the letters from the queue, one at a time, and prints these on the screen. The constructor takes the shared `Queue` as argument, as well as a parameter specifying the interval between two successive `read(...)`.

```
1  class Consumer implements Runnable{
2
3          private Queue theQueue;
4          private int interval;
5
6          public Consumer(Queue q, int inter){
7                  theQueue = q;
8                  interval = inter;
9          }
10
11         public void run(){
12                 try{
13                         while(true){
14                                 String s = (theQueue.read()).toString();
15                                 System.out.println("Consumer: read() returned ,"+
16                                                    "got character: "+s);
17                                 Thread.sleep(interval);
18                         }
19                 }catch(InterruptedException e ){
20                         // Exeception Handling
21                 }
22         }
23 }
```

Figure 2.18: The Consumer-class for the Producer-Consumer example.

Both the `Producer` and `Consumer` permits a configurable interval between successive `read()` and `write(...)` operations so as to enable illustrating the behavior when either is faster than the other.

```
1  class ProducerConsumer{
2
3          public static void main(String[] args){
4                  Queue sharedQueue = new Queue(5);
5                  Thread p = new Thread(new Producer(sharedQueue, 100));
6                  Thread c = new Thread(new Consumer(sharedQueue, 500));
7                  p.start();
8                  c.start();
9          }
10 }
```

Figure 2.19: The test-class with `public static void main(...)`.

The test-class for the Producer-Consumer example is illustrated in figure 2.19, and simply creates the shared `Queue` and starts a thread for each of the `Producer` and

`Consumer`.

Remaining is the `Queue`-class, which must ensure that data are not written when the queue is full, and that data are not read when the queue is empty. The `Queue`-class is illustrated in figure 2.20.

```
1  class Queue{
2          private int toRead = 0;
3          private Object[] content;
4
5          private int head, tail, capacity, count;
6
7          public Queue (int cap){
8                  content = new Object[cap];
9                  capacity = cap;  head=0; tail=0; count=0;
10         }
11
12         public synchronized void write(Object o) throws InterruptedException{
13                 while (count == capacity){
14                         System.out.println("Producer: queue full when write("
15                                         +(o.toString())+")");
16                         wait();
17                 }
18                 content[tail]=o;
19                 count = count +1;
20                 tail = (tail + 1) % capacity;
21                 System.out.println("Producer: write("+(o.toString())+") succeded");
22                 notifyAll();
23         }
24
25         public synchronized Object read() throws InterruptedException{
26                 while (count == 0){
27                         System.out.println("Consumer: Queue empty when read()");
28                         wait();
29                 }
30                 Object ret = content[head];
31                 content[head] = null;
32                 count = count -1;
33                 head = (head + 1) % capacity;
34                 notifyAll();
35                 return ret;
36         }
37 }
```

Figure 2.20: The `Queue`-class, using `synchronized`, `wait()` and `notifyAll()`, supporting concurrent access.

Concurrency-wise, the `Queue`-class in figure 2.20 is identical to the alternating writers example in figure 2.16: upon a thread (producer or consumer) acquiring the MUTEX, the first task is to verify if the conditions for successful completion the operation (`read()` and `write(...)` on the `Queue`) are satisfied – and, if not, and execute `wait()`. Upon

having successfully completed an operation modifying the state of the monitor (added or removed elements from the queue), `notifyAll()` is invoked to awaken all suspended threads so that they may verify anew if the conditions for successful completion of their desired operation are satisfied.

Contrary to the alternating writers example in figure 2.16, this producer-consumer example illustrates how `InterruptedExceptions`, generated by `wait()` are propagated from inside the `Queue` where they may be generated by way of a `wait()` being interrupted, and to the actual producer and consumer.

Executing the program in figure 2.19 with the producer being "faster" than the consumer (*i.e.,* with shorter interval between two `write(...)` than between two `read()` produces something like the below::

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java ProducerConsumer
Producer: write(a) succeded
Consumer: read() returned, got character: a
Producer: write(b) succeded
Producer: write(c) succeded
Producer: write(d) succeded
Producer: write(e) succeded
Consumer: read() returned, got character: b
Producer: write(f) succeded
Producer: write(g) succeded
Producer: queue full when write(h)
Consumer: read() returned, got character: c
Producer: write(h) succeded
Producer: queue full when write(i)
```

Executing the program in figure 2.19 with the consumer being "faster" than the producer (*i.e.,* with longer interval between two `write(...)` than between two `read()` produces something like the below::

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java ProducerConsumer
Producer: write(a) succeded
Consumer: read() returned, got character: a
Consumer: Queue empty when read()
Producer: write(b) succeded
Consumer: read() returned, got character: b
Consumer: Queue empty when read()
Producer: write(c) succeded
Consumer: read() returned, got character: c
Consumer: Queue empty when read()
Producer: write(d) succeded
Consumer: read() returned, got character: d
Consumer: Queue empty when read()
```

The Producer-Consumer problem is, commonly, encountered in pipelining architectures, for asynchronous communication between active objects, etc.

2.10 Other Synchronization mechanisms in Java

The basic, and only, synchronization mechanism in Java is the Monitor. Semaphores, MUTEXes etc. can be constructed by way of the Monitor – if needed. While it is generally recommended to carefully consider proper encapsulation of data in Java, and rigorous use of `private` (for attributes) and `synchronized` (for methods), this section illustrates how a simple MUTEX can be implemented in Java, as well as the (limited) ability to provide atomic access to (certain) variables by way of the `volatile` keyword.

2.10.1 Simple MUTEX in Java

Declaring and using a MUTEX in Java is illustrated in figure 2.21.

```
1
2         //.....
3         Object myMutex = new Object();
4
5         // Outside Critical Section ....
6
7         synchronized (myMutex){
8                 // Critical Section Goes Here
9         }
10
11        // Back outside the Critical Section ....
```

Figure 2.21: Abusing a Java Monitor as a simple MUTEX.

Note that `synchronized` assumes an object.

2.10.2 Volatile Variables

In Java, it is possible to declare a variable as `volatile`, as illustrated in figure 2.22.

```
1
2         volatile int i;
```

Figure 2.22: The `volatile` keyword.

A variable declared volatile means, that operations on that variable are *atomic – i.e.,* indivisible. The value of a variable declared `volatile` will never be cached thread-locally, rather : all reads and writes will go straight to "main memory", thus a thread reading the variable will always get the most recent value written. Access to the variable acts as-if it is enclosed in a synchronized block, synchronized on itself. The operational words here are act as-if, as there is no actual MUTEX involved, and this entails that:

- Any sequence of operations on the variable, even by the same thread, are not guaranteed to be indivisible.
- The `wait()`, `notify()` and `notifyAll()` operations do not apply.

In general, `synchronized` and monitors offers more flexibility than does declaring variables `volatile`, and a monitor permits rendering a sequence of operations on the same data (in the monitor) indivisible – however necessitate the creation of an Object.

Also, `volatile` does not mean atomic. Consider what the value of the following code would be, assuming that the two threads execute to completion:

```
1
2         volatile int i = 0;
3
4         // Thread a:
5                 i++;
6
7         // Thread b:
8                 i--;
```

The intuitive answer should be that `i == 0` – however, that is not so. The operations `i++` and `i--` are actually compound operations (*e.g.,* `i = i + 1`), thus:

1. read `i` to a local variable;
2. increment (or decrement) that local variable
3. write `i` to the `volatile` memory location

Therefore, the above code in reality becomes:

```
1
2         volatile int i = 0;
3
4         // Thread a:
5                 local_a = i
6                 local_a = local_a + 1
7                 i = local_a
8
9         // Thread b:
10                local_b = i
11                local_b = local_b - 1
12                i = local_b
```

It may be that the scheduling be such that the order of execution of these instructions is:

```
1
2         volatile int i = 0;
3
4                 local_a = i
5                 local_b = i
6                 local_a = local_a + 1
7                 local_b = local_b - 1
8                 i = local_a
9                 i = local_b
```

In which case, `i == -1`.

Or, scheduling may cause an execution order thus:

```
1
2         volatile int i = 0;
3
4                 local_a = i
5                 local_b = i
6                 local_a = local_a + 1
7                 local_b = local_b - 1
8                 i = local_b
9                 i = local_a
```

In which case, `i == 1`.

Thus, the `volatile` keyword is to use with much caution. In most cases, proper use of `synchronized` is a better – and clearer – solution to concurrency management.

2.10.3 Stopping a Thread as an example of using volatile

Volatile variables can be used as an alternative approach to the use of `interrupt()` (section 2.6) for requesting that a running thread be gracefully terminated. Figure 2.23 illustrates the basic idea for stopping a thread in this fashion.

A private boolean variable `doStop` indicates if the thread has requested to stop or not *at the earliest convenience of the thread*. This variable must be private, and initially false. A method `public void kindlyStop()` is used to, from "outside the thread", request that the thread stops by way of setting the variable `stop` to `true`. This is a request only, as the thread will execute according to the `run()` method – which should occasionally, and when any data is in a consistent state, verify if the thread has been requested to terminate.

The `run()` method contains a `while (!stop)` loop, essentially letting the thread run until the `stop` variable becomes true, similar to the example in section 2.6. At the end of each iteration of the `while(...)` loop, care should be taken that shared data are left in a consistent state, in case termination of the thread is requested.

```
1  class ThreadToStop extends Thread{
2
3          private volatile boolean stop;
4
5          ThreadToStop (){
6                  stop = false;
7                  start();
8          }
9
10         public void kindlyStop(){
11                 stop = true;
12         }
13
14         public void run(){
15                 while (!stop){
16                         // Do Whatever the Thread is supposed to do
17
18                         // Ensure that shared data are in a consistent state and thus
19                         // it is safe to stop, if so requested
20                 }
21         }
22 }
```

Figure 2.23: A basic mechanism for requesting that a running thread be terminated using simple method invocation.

Once `stop` becomes true, `run()` returns and the thread is stopped.

The variable `stop` is declared to be `volatile` – meaning that all access to that variable is *atomic*. Thus, no two threads can access the variable at the same time – much as if access to this variable had been protected by a MUTEX.

Figure 2.24 illustrates how this basic mechanism is used for requesting that a thread *kindly stops*: the thread `t1` is created, after which the main thread of the program sleeps for 150ms (see section 2.5). During those 150ms `t1` executes. Once the main thread wakes up, it invokes `t1.kindlyStop()`, requesting that `t1` terminates at its next convenience.

```
1  class StopThreadExample{
2
3          public static void main(String[] args){
4                  ThreadToStop t1 = new ThreadToStop();
5                  try{
6                          Thread.sleep(150);
7                          t1.kindlyStop();
8                  }catch (InterruptedException e){
9                          // handle if were interrupted while sleeping
10                 }
11         }
12 }
```

Figure 2.24: A basic mechanism for requesting that a running thread be terminated using simple method invocation.

Figure 2.25 illustrates a complete program, using the principles from figure 2.24: a thread is created which simply outputs "a" until it is stopped. The main thread sleeps for 150ms, then request that the created thread stops.

```
1  class ThreadToStop extends Thread{
2
3          private volatile boolean stop;
4          private String s;
5          ThreadToStop (String t){
6                  stop = false;
7                  s=t;
8                  start();
9          }
10
11         public void kindlyStop(){
12                 stop = true;
13
14         }
15
16         public void run(){
17
18                 while (!stop){
19                         System.out.print(s);
20                 }
21         }
22 }
23 class StopThreadExample{
24
25         public static void main(String[] args){
26                 ThreadToStop t1 = new ThreadToStop("a");
27                 try{
28                         Thread.sleep(150);
29                         t1.kindlyStop();
30                 }catch (InterruptedException e){
31                         // handle if were interrupted while sleeping
32                 }
33         }
34 }
```

Figure 2.25: An example program, illustrating requesting a thread to terminate

2.11 Scheduling Future and Recurrent Events

Java offers a facility for scheduling tasks for future execution in a background thread by way of the classes `java.util.Timer` and `java.util.TimerTask`. Such tasks may be scheduled for one-time execution, or for repeated execution at regular intervals. Figure 2.26 illustrates how this facility is used for scheduling a simple task (printing `Hello World`) 5 seconds after the program is executed.

Executing the program in figure 2.27 produces something like the below:

```
1  import java.util.Timer;
2  import java.util.TimerTask;
3
4  public class TimerTaskTest{
5
6      Timer timer;
7
8      public TimerTaskTest(){
9          timer = new Timer();
10         timer.schedule(new DelayedTask(), 5000);
11         }
12
13     class DelayedTask extends TimerTask{
14         public void run(){
15             System.out.println("Hello World");
16             timer.cancel(); //Terminate the timer thread
17         }
18     }
19
20     public static void main(String[] args){
21         new TimerTaskTest();
22     }
23 }
```

Figure 2.26: A simple `java.util.Timer` scheduling a task for future execution.

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java TimerTaskTest
Hello World
```

`Timer` is a class which, when instantiated, creates *one* thread used for executing all the `TimerTask`s scheduled for it by way of the `schedule(...)` method.

`TimerTask` is an *abstract class*, specified to `implements Runnable` – hence, the task to execute must be specified in the `run()` method of a class which `extends TimerTask`. The `run()` method contains, just as when creating a regular `Thread`, the instructions to execute when the `TimerTask` is scheduled.

Scheduling a `TimerTask` for execution consists of executing `schedule(...)` on the appropriate `Timer` object, giving as argument `TimerTask` object, as well as other scheduling parameters such as the delay until first execution of the task, or the interval between subsequent executions.

This is illustrated in line 10, where a `DelayedTask` is scheduled for execution after 5000 ms.

As a Java program continues to run for as long as there are threads (including those created from a `Timer`) running, line 11 explicitly calls `timer.cancel()`, indication that this specific thread is terminated.

Scheduling recurrent tasks is as simple as scheduling future tasks, as illustrated in fig-

ure 2.27. This program is identical to the one in figure 2.26, except for line 10, where the second argument to `schedule(....)` is the "time until the first execution of this task" as in the example in figure 2.26, and the third argument is the "interval between two executions of this task".

```
1  import java.util.Timer;
2  import java.util.TimerTask;
3
4  public class TimerTaskTest{
5
6      Timer timer;
7
8      public TimerTaskTest(){
9          timer = new Timer();
10         timer.schedule(new RecurrentTask(), 0, 1000);
11         }
12
13     class RecurrentTask extends TimerTask{
14         public void run() {
15             System.out.println("Hello World");
16             timer.cancel(); //Terminate the timer thread
17         }
18     }
19
20     public static void main(String[] args[]){
21         new TimerTaskTest();
22     }
23 }
```

Figure 2.27: A simple `java.util.Timer` scheduling a task for periodic execution every 1000 ms.

Executing the program in figure 2.27 produces something like the below:

```
Thomas-Heide-Clausens-iMac:Test Code voop$ java TimerTaskTest
Hello World
Hello World
Hello World
....
```

Caveat lector: as one thread is created per `Timer`, all `TimerTasks` scheduled for the same `Timer` share this. Hence, if a `TimerTask` in its `run()` method has *e.g.,* an infinite loop, other `TimerTasks` scheduled for the same `Timer` may never be executed.

2.11.1 Daemon Tasks

In the examples in figure 2.26 and in figure 2.27, explicit calls to `timer.cancel()` were required, least the program would not terminate. In some cases, a program may contain a set of *periodic background* `TimerTasks` (*e.g.,* periodically sending a "I am here" message on a network). In order for the program to terminate, such `TimerTasks` would have to be independently canceled by way of their `run()` methods executing `timer.cancel()`. `TimerTask`'s are threads, and as seen in section 2.6, signaling to a thread that it should terminate is not trivial. To overcome this, Java supports *Daemon Tasks*.

A Daemon task is a `TimerTask` which has as characteristics that if the only threads left in a program are such Daemon tasks, the program exits.

A task is created as a Daemon Task by the associated `Timer` being instantiated thus:

```
1        timer = new Timer(true);
```

A `TimerTask` subsequently scheduled by invoking `schedule(...)` on this `Timer` will execute only as long as there are other (non-daemon) threads running in the program.

2.11.2 Caveat: No Real-Time Guarantees

`Timer` does not offer real-time guarantees. The interval before a `TimerTask` is executed, or between two executions of a recurrent `TimerTask` is dependent on scheduling, on the thread associated with a `TimerTask` being interrupted etc. The execution schedule is "mostly accurate", but not guaranteed to be precise.

Chapter 3

A Primer On: Stream-Based Input/Output

One of the two ways, in which Java supports input and output operations, including for sending and receiving data over the network and reading files, is by way of I/O-streams. An I/O stream represents a source or a sink for data – *i.e.,* something from which data can be read, or to which data can be written.

This chapter provides a brief introduction to the principles of stream-based I/O in Java. The alternative way in which Java supports input and output operations is through I/O-channels - which will be described in chapter 4.

Chapter **??** illustrates how these two I/O concepts (streams and channels) are used for reading and writing a file from disk, and chapter **??** how they are used for network programming.

3.1 Basic Stream-Based Input/Output Principles

The Java class `java.io.InputStream` and `java.io.OutputStream` are abstract classes, at the base for all Java I/O streams. As everything in Java – ints, Strings, Objects etc. – can be represented as octets (bytes), `java.io.InputStream` and `java.io.OutputStream` are basic byte-streams, and operations on these are reading and writing "raw" bytes.

Being abstract classes, `java.io.InputStream` and `java.io.OutputStream` must be extended (subclassed) in order to be used. All classes, extending `java.io.InputStream` must provide a method `read()`, which returns the next byte available (and -1 if there are none), and all classes extending `java.io.OutputStream` must provide a method `write(int b)` for writing the byte *b*. Most classes extending `java.io.InputStream` and `java.io.OutputStream` provide more appropriate mechanisms for reading/writing

other data types, as will be explained later.

Almost all operations on `java.io.InputStream`, `java.io.OutputStream` and their subclasses may throw exceptions (`java.IO.IOException` and subclasses), which must be caught and handled.

3.1.1 `java.io.InputStream`

The piece of Java code in figure 3.1 illustrates the basic way, in which to read octets from a `java.io.InputStream` until that stream is empty.

```
1        InputStream in = new ..... // Some subclass of InputStream
2
3        int read = in.read();
4        while (read != -1){
5                // Do that which needs to be done with the read octet
6                read=in.read();
7        }
8        in.close();
```

Figure 3.1: Reading octets, one-by-one, from `java.io.InputStream`.

`java.io.InputStream` also provides a method for reading a sequence of bytes into an array, by way of the method `int read(byte[] b)`. The byte-array must be created before calling this method, as illustrated in figure 3.2, and the `int` returned is the number of bytes read from the stream.

```
1        InputStream in = new ..... // Some subclass of InputStream
2        byte[] b = new byte[42]
3
4        int read = in.read(b);
5        in.close();
```

Figure 3.2: Reading a sequence of octets at once, from `java.io.InputStream`.

Of note in both of these examples is, that the last instruction is `in.close()`. In general, all streams (input and output, both) must be explicitly closed, least the system may be left in an inconsistent state. Technically, `close()` releases any and all system resources associated with the stream (frees buffers, de-allocates memory, ...)

Another, more general, way of reading a sequence of bytes from an `java.io.InputStream` is by way of `int read(byte[] b, int offset, int length)`. This method reads bytes from the `java.io.InputStream` and into the array `b`. The first byte read from the stream is inserted into `b[offset]`, the next into `b[offset+1]` etc. The parameter `length` specifies the maximum number of bytes that should be read from the stream.

The `int` returned is the number of bytes read, or `-1` in case the end of the stream is reached. This is illustrated in figure 3.3.

```
1        InputStream in = new ..... // Some subclass of InputStream
2        byte[] b = new byte[42]
3
4        int read = in.read(b, 4, 8);
5        in.close();
```

Figure 3.3: Reading a sequence of 8 octets, starting at index 4 at once, from `java.io.InputStream`.

Note that `int read(byte[] b, int offset, int length)` works, internally, by repeated calls to `read()`.

3.1.2 `java.io.OutputStream`

The piece of Java code in figure 3.4 illustrates the basic way, in which to write an octet to an `java.io.OutputStream`.

```
1        OutputStream out = new ..... // Some subclass of OutputStream
2
3        byte b = 42;
4        out.write(b);
5
6        out.flush();
7        out.close();
```

Figure 3.4: Writing a single octet to `java.io.OutputStream`.

Of note in this examples is, that before calling `out.close()`, a call is made to `out.flush()`. Calling this method is primordial: data written via an `java.IO.OutputStream` to (for example) a network connection may not be immediately sent across the network, but stored in a buffer - for example, waiting for a TCP connection to be ready to receive data. As calling `close()` immediately frees system resources associated with the stream, including freeing buffers, such data may not have actually been sent before the buffer in which they were held were freed.

`java.io.OutputStream` also provides a method for writing a sequence of bytes from an array onto the stream, by way of the method `void write(byte[] b)`, as illustrated in figure 3.5.

Analog to `java.io.InputStream`, another, more general, way of writing bytes to a `java.io.OutputStream` is by way of `void write(byte[] b, int offset, int length)`,

```
1        OutputStream out = new ..... // Some subclass of OutputStream
2        byte[] b = "Hello World".getBytes();
3
4        out.write(b);
5
6        out.flush();
7        out.close();
```

Figure 3.5: Writing an array of octets at once, from `java.io.InputStream`.

where `offset` specifies the index of the first byte to write, and `length` the number of bytes from `b` that should be written.

3.2 Buffered Streams

It is often not particularly efficient to use "raw" byte-streams directly. For example, when accessing a file on a disk, or accessing data across the network, byte-by-byte access is inefficient: each `read()` and `write()` translates into a call to the underlying operating system to acquire the next byte – which may trigger disk or network activity.

A buffered stream works on a chunk of memory, from which data is read or to which it is written. Only when the buffer is full (when writing) or empty (when reading) is a call made to the underlying operating system. This permits, for example, bulk-reads of chunks of multiple octets from the disk at once, rather than causing separate reads for each byte – and permits sending one large packet containing many bytes across a network, rather than many small packets, each with just a single byte.

Converting an unbuffered stream into a buffered stream is relatively easy, illustrated in figure 3.6 and figure 3.7.

```
1
2        OutputStream out = new ..... // Some subclass of OutputStream
3        BufferedOutputStream bout = new BufferedOutputStream(out);
4
5        InputStream in = new ..... // Some subclass of InputStream
6        BufferedInputStream bin = new BufferedInputStream(bin);
```

Figure 3.6: Converting an unbuffered stream into a buffered stream with default buffer size.

It is worth noting that the default buffer size, used by Java when no explicit buffer size is provided, varies both with platform (Windows, MacOS, ...) and with Java version.

Reading and writing from these buffered streams is analog to their unbuffered, raw, counterparts.

```
1
2           OutputStream out = new ..... // Some subclass of OutputStream
3           BufferedOutputStream bout = new BufferedOutputStream(out, 8 * 1024);
4
5           InputStream in = new ..... // Some subclass of InputStream
6           BufferedInputStream bin = new BufferedInputStream(bin, 8*1024);
```

Figure 3.7: Converting an unbuffered stream into a buffered stream, specifying the size of the buffer, in bytes. In this example 8 kB buffers are allocated.

`BufferedOutputStream` provides the methods for writing:

- `void write(byte[] b, int offset, int length)`
- `void write(int b)`

Figure 3.8 illustrates writing bytes to a `BufferedOutputStream`

```
1           OutputStream out = new ..... // Some subclass of OutputStream
2           BufferedOutputStream bout = new BufferedOutputStream(out, 8 * 1024);
3           bout.write("Hello World".getBytes());
4           bout.flush();
5           bout.close();
6           out.flush();
7           out.close();
```

Figure 3.8: Writing to a buffered stream, created with 8 kB buffers allocated.

Of note in this examples is, that before calling `bout.close()`, a call is made to `bout.flush()` – for the reasons of ensuring that any buffered data are written prior to closing the buffered stream. Also of note is that after closing the `BufferedOutputStream`, `out.flush()` and `out.close()` are called. While this is not strictly necessary – closing a `BufferedOutputStream` also closes the related `OutputStream`, it is good practice to always close any streams that have been opened.

`BufferedInputStream` provides the methods for reading:

- `int read(byte[] b, int offset, int length)`, as in the unbuffered `InputStream`
- `int read()`, as in the unbuffered `InputStream`

Figure 3.9 illustrates reading bytes from a `BufferedInputStream`

As with its unbuffered counterparts, a `java.io.BoundedInputStream` is closed using `close()` – and, while not strictly necessary, the good practice of closing all streams that have been opened is respected, by successively calling `bin.close()` and `in.close()`.

```
1
2         InputStream in = new ..... // Some subclass of OutputStream
3         BufferedInputStream bin = new BufferedInputStream(out, 8 * 1024);
4         byte[] b = new byte[42];
5         int read = in.read(b);
6         bin.close();
7         in.close();
```

Figure 3.9: Reading from a buffered stream, created with 8kB of buffers allocated.

3.3 Readers and Writers: Character-based I/O

Java provides explicit support for character-based I/O, such as reading or writing a text file. Text is composed of characters, and characters in Java are associated with a character set, permitting *e.g.,* national characters such as æ ø and å to be properly represented. This support is manifested by way of a family of classes called Readers and Writers. Both `java.io.Reader` and `java.io.Writer` are, analog to `java.io.InputStream` and `java.io.OutputStream`, abstract classes.

All `java.io.Reader`'s and `java.io.Writer`'s must be properly closed (both) and flushed (`java.io.Writer`) by way of `close()` and `flush()` - for the same reasons as given for streams.

Almost all operations on `java.io.Reader`, `java.io.Writer` and their subclasses may throw exceptions (`java.IO.IOException` and subclasses), which must be caught and handled. As Readers and Writers also provide character conversion, they may also throw a `java.io.UnsupportedEncodingException`, which must be caught and handled.

3.3.1 InputStreamReader and OutputStreamWriter

An `java.io.InputStreamReader` is a subclass of `java.io.Reader`, providing a bridge from a byte stream to a character stream – specifically reading bytes from a `java.io.InputStream` and decoding these into characters by way of a specified character set. Similarly, a `java.io.OutputStreamWriter` is a subclass of `java.io.Writer`, providing a bridge from a character stream to a byte stream – specifically, converting characters to bytes by way of a specified character set, and writing these to a `java.io.OutputStream`.

The piece of Java code in figure 3.10 illustrates the basic way, in which to read characters from a `java.io.InputStreamReader` until the stream associated with that reader is empty.

The method `read()` returns an integer, which can be converted to a character by way of the *typecast* `(char)`. Thus, `char c = (char) read;` returns the next character, resulting from reading and decoding the `java.io.InputStream`.

```
1         InputStream in = new ..... // Some subclass of InputStream
2         Reader r = new InputStreamReader(in);
3         int read = r.read();
4         while (read != -1){
5                 char c = (char) read;
6                 // Do that which needs to be done with the read character
7                 read=r.read();
8         }
9         in.close();
```

Figure 3.10: Reading octets, one-by-one, from `java.io.InputStreamReader`.

`java.io.InputStreamReader` also provides a method for reading a sequence of characters into an array, by way of the method `int read(char[] v, int offset, int length)`. The char-array must be created before calling this method, as illustrated in figure 3.11, and the `int` returned is the number of bytes read from the stream.

```
1         InputStream in = new ..... // Some subclass of InputStream
2         Reader r = new InputStreamReader(in);
3         byte[] c = new char[42]
4
5         int read = r.read(c, 4, 8);
6         r.close();
```

Figure 3.11: Reading a sequence of 8 characters at once, starting at index 4, from `java.io.InputStreamReader`.

In an analog way, the piece of Java code in figure 3.12 illustrates the basic way, in which to write a single character to a `java.io.OutputStreamWriter`.

```
1         OutputStream out = new ..... // Some subclass of OutputStream
2         Writer w = new OutputStreamWriter(out);
3         char c = 'a';
4         w.write(c);
5
6         w.flush();
7         w.close();
```

Figure 3.12: Writing a single character to `java.io.OutputStreamWriter`.

`java.io.OutputStreamWriter` also provides a method for writing a sequence of characters from an array onto the stream, by way of the method `void write(char[] v, int offset, int length)`, where `offset` specifies the index of the first byte to write, and `length` the number of bytes from `b` that should be written, as illustrated in figure 3.13.

The specificity of `java.io.Reader` and `java.io.Writer` is, that they are able to au-

```
1        OutputStream out = new ..... // Some subclass of OutputStream
2        Writer w = new OutputStreamWriter(out);
3        char[] c = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};
4
5        w.write(c, 0, 4);
6
7        w.flush();
8        w.close();
```

Figure 3.13: Writing a sequence of 4 characters at once from the array c, starting at index 0, to a `java.io.OutputStreamWriter`.

tomatically perform character set conversion. Internally, Java represents characters as UTF-8 encoded – whereas files on a disk may be, for example, ASCII or 8859_1. In order to automatically convert a stream from *e.g.,* 8859_1 to UTF-8 when read or write, the constructor of the `java.io.InputStreamReader` and `java.io.OutputStreamWriter` can take the character encoding as argument, as illustrated in figure **??**.

```
1        OutputStream out = new ..... // Some subclass of OutputStream
2        Writer w = new OutputStreamWriter(out, "8859_1");
3
4        // ....
5
6        InputStream in = new ..... // Some subclass of InputStream
7        Reader r = new InputStreamReader(in, "8859_1");
```

Figure 3.14: Selecting text encoding in the constructor of `java.io.OutputStreamWriter` and `java.io.InputStreamReader`.

Note that, unfortunately, Java doesn't provide any way of accessing a list of all character encodings that are supported, nor for a program to install its own character encoding.

3.3.2 BufferedReader and BufferedWriter

As with streams, it is often not particularly efficient to use "raw" character-readers/writers directly. For example, when accessing a file on a disk, or accessing data across the network, character-by-character access is as inefficient as is byte-by-byte access for a stream: each `read()` and `write()` translates into a call to the underlying operating system to acquire the next byte – which may trigger disk or network activity.

As with buffered streams, a buffered reader/writer works on a chunk of memory, from which data is read or to which it is written. Only when the buffer is full (when writing) or empty (when reading) is a call made to the underlying operating system.

Converting an unbuffered reader/writer into a buffered reader/writer is relatively easy, illustrated in figure 3.15. The constructors for `java.io.BufferedReader` and `java.io.BufferedWriter` permit – as in the case of `java.io.BufferedInputStream` and `java.io.BufferedOutputStream` - specifying the buffer size, illustrated in figure 3.16.

```
1         InputStream in = new ..... // Some subclass of InputStream
2         Reader r = new InputStreamReader(in);
3         BufferedReader br = new BufferedReader(r);
4                 //       Read from br
5         br.close();
6         r.close();
7         in.close();
8
9         OutputStream out = new ..... // Some subclass of OutputStream
10        Writer w = new OutputStreamWriter(out);
11        BufferedWriter bw = new BufferedWriter(w);
12                //       Write to bw
13        bw.flush();
14        bw.close()
15        out.close();
```

Figure 3.15: Converting a `java.io.Reader` and `java.io.Writer` to `java.io.BufferedReader` and `java.io.BufferedWriter`, respectively.

`java.io.BufferedReader` supports the same methods for reading characters as its unbuffered counterpart,*i.e.,* `read()` and `int read(char[] v, int offset, int length)`. Furthermore, `java.io.BufferedReader` provides `String readLine()`, which reads and returns a line of text – *i.e.,* a sequence of chargers terminated by either of *line feed* `'\n'`, *carriage return* `'\r'` or *carriage-return-line-feed* `"\r\n"`. These are illustrated in figure 3.17.

`java.io.BufferedWriter` supports the same methods for writing characters as its unbuffered counterpart,*i.e.,* `void write(char c)` and `void write(char[] v, int offset, int length)`. Furthermore, `java.io.BufferedWriter` provides `void write(String s, int off, int len)`, which writes a portion of the string, `s`, delimited by `off` and `len`. Finally, `java.io.BufferedWriter` provides `void newLine()`, which writes the line separator – *i.e.,* either of *line feed* `'\n'`, *carriage return* `'\r'` or *carriage-return-line-feed* `"\r\n"`[1] These are illustrated in figure 3.18.

[1]Which exact symbol is used as line separator is, unfortunately, system dependent and may not be *line feed* `'\n'`. The actual line separator used is defined in the system property `line.seperator`, can be set by way of `System.setProperty("line.separator","\r\n")` and inspected by way of `System.setProperty("line.separator")`.

```
1          InputStream in = new ..... // Some subclass of InputStream
2          Reader r = new InputStreamReader(in);
3          BufferedReader br = new BufferedReader(r, 8 * 1024);
4                  //       Read from br
5          br.close();
6          r.close();
7          in.close();
8
9          OutputStream out = new ..... // Some subclass of OutputStream
10         Writer w = new OutputStreamWriter(out);
11         BufferedWriter bw = new BufferedWriter(w, 8 * 1024);
12                 //       Write to bw
13         bw.flush();
14         bw.close()
15         out.close();
```

Figure 3.16: Converting a `java.io.Reader` and `java.io.Writer` to `java.io.BufferedReader` and `java.io.BufferedWriter`, respectively, with explicit buffer sizes given.

3.3.3 Caveat for PrintWriter

A class exist, `java.io.PrintWriter` which does much the same as `java.io.BufferedWriter` in that it permits writing a complete line at a time. `java.io.PrintWriter` also provides various methods such as `print(int i)`, doing much of the conversion from primitive data types into `String`, as well as symmetric methods `println(....)` which terminate each written line by the symbol(s) recorded in the system property `line.seperator`.

`java.io.PrintWriter` is, however, to be vary of, in that methods from `java.io.PrintWriter` never throw any exceptions (some of its constructors may, however), but rather provides methods to inquire if an error has occurred. This alone makes it a "dangerous" class to use, in that its exception behavior radically differ from other Java classes – essentially, it does "Lazy Exception Handling" as described in section 1.9, and should only be used in case there is no way of making use of `java.io.BufferedWriter` in its place.

3.4 `System.out`, `System.err`, and `System.in`

Java provides a class `System`, which encapsulates the interaction between a Java program and the system. This class cannot be instantiated, and all methods and attributes are `static`. This chapter has already, in passing and without further explanation, introduced `System.setProperty("line.separator")`. The most commonly used part functionality from that class is, however, reading input from the keyboard and writing output to the screen by way of `System.out.println(....)`. This section will explore this system-provided I/O stream, as well as its counterparts `System.err` and `System.in`.

```
1   InputStream in = new ..... // Some subclass of InputStream
2   Reader r = new InputStreamReader(in);
3   BufferedReader br = new BufferedReader(r);
4
5   char[] c = new char[5];
6
7   int read = br.read();           //      Read single character
8   char c = (char) read;
9
10  read = br.read(c, 0, 4);        //      Read 4 charters into array c
11
12  String s = br.readLine();       //      Read a string, terminated by \r \n or \r\n
13
14  br.close();
15  r.close();
16  in.close();
```

Figure 3.17: Different ways of reading from a `java.io.BufferedReader`.

These streams are known as "stdout" (or "standard-out"), "stdin" (or "standard-in") and "stderr" (or "standard-error").

The naming and details of stderr and stdout is, largely, historical: they exist so as to provide the ability - for example - when executing a program to direct all error messages to a dedicated log-file, and all normal output to a text file. In Unix, using the BASH shell, this could look something like this:

```
sh-3.2$ ./myprogram 1>programoutput 2>errorlog
```

To support this, Java provides both `System.out` and `System.err` – with both being static instances of `java.io.PrintWriter`, therefore have the various `print(...)` and `println(....)` methods, illustrated in figure 3.19.

`System.in` is a static instance of a `java.io.InputStream` and permits reading from the standard input stream. By default, this is the keyboard, and thus reading from this stream permits reading user input. An example is shown in figure 3.20.

Recall that a `java.io.InputStream` simply supplies "raw" octets, one-by-one, and the desired input is characters. Line 4, therefore, wraps `System.in` into a `java.io.InputStreamReader` which – as all readers – convert bytes to characters. An `java.io.InputStreamReader` returns bytes one-by-one – and is, therefore, in line 5 in figure 3.20, wrapped in a `java.io.BufferedReader` so as to provide a method `String readLine()` – used in line 7 in figure 3.20.

```
1   OutputStream out = new ..... // Some subclass of OutputStream
2   Writer w = new OutputStreamWriter(out);
3   BufferedWriter bw = new BufferedWriter(w);
4
5   char c = 'a';
6   w.write(c);
7
8   char[] c = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};
9   w.write(c, 0, 4);
10
11  w.newLine();
12
13  String s = "Hello World";
14  w.write(s, 0, s.length());
15
16  bw.flush();
17  bw.close();
18  w.flush();
19  w.close()
20  out.flush();
21  out.close();
```

Figure 3.18: Different ways of writing to a `java.io.BufferedWriter`.

```
1   class HelloWorld{
2       public static void main(String[] args){
3           System.out.println("Hello World to stdout");
4           System.err.println("Hello World to stderr");
5       }
6   }
```

Figure 3.19: Simple example of writing to `System.out` and `System.err`.

```
1   import java.io.*;
2   class Echo{
3       public static void main(String[] args) throws Exception{
4           InputStreamReader isr = new InputStreamReader(System.in);
5           BufferedReader br = new BufferedReader(isr);
6           System.out.print("Type Something: ");
7           String s = br.readLine();
8           System.out.println("You typed: "+s);
9           br.close();
10      }
11  }
```

Figure 3.20: Reading from the default stdin - the keyboard.

Chapter 4

A Primer On: Channel-Based Input/Output

The second way in which Java supports input and output operations, including for sending and receiving data over the network and reading files, is by way of I/O-channels.

The main conceptual difference between channels and streams is that channels are designed to support *non-blocking* I/O-operations. In contrast, the API defines the `read`-method of an `InputStream` to be always blocking. Thus, if method `in.read()` is invoked on an `InputStream in` and the next byte of stream `in` is not yet available (*e.g.*, because the next packet over the network was not yet received), the `read`-method loops until this byte is available and returns only then.

4.1 Buffers

Buffers are used as endpoints of channels and extend the abstract class `java.nio.Buffer`. Buffers are used to read from and write to channels.

The `Buffer` class itself does not specify methods for data transfer, *i.e.*, reading from and writing to a buffer. These have to be defined in subclasses, like the abstract class `java.nio.ByteBuffer`, which defines the (abstract) methods `get` and `put`.

A buffer can be either read-only or writable. A `Buffer buf` is writable if and only if `buf.isReadOnly()` is `false`. Trying to write to a read-only buffer results in a `ReadOnlyBufferException` being thrown.

4.1.1 Indexing

A buffer is a container for data and can be thought of as essentially being an array, with a few extras. There are two possbilities of indexing inside a buffer: relative and absolute indexing.

Three values determine indexing: *position*, *limit*, and *capacity*. Valid indices for data transfer operations on the buffer are integers between 0 and *limit* − 1. The three values can be read via the methods `position()`, `limit()`, and `capacity()`. The value *capacity* is fixed upon creation of the `Buffer` object and cannot be changed. The other two can be set with the methods `position(int newPosition)` and `limit(int newLimit)`.

The same `Buffer` object can be used more than once. That's why *limit* is not necessarily equal to *capacity*, but only less or equal: The value *limit* denotes the size of the current piece of data in the buffer. If the current data is no longer needed to reside in the buffer, new data can be put into the buffer and *limit* can be updated according to the new data size.

Absolute indexing is used when invoking the methods `get(int index)` or `put(int index, byte b)`, which can both throw an `IndexOutOfBoundsException`. The given integer `index` has to be between 0 and *limit* − 1, and is the absolute index inside the buffer of the byte read resp. written. Figure 4.1 shows an example of reading the contents of a buffer using absolute indexing.

```
1       ByteBuffer buf = ... // initialize buffer
2
3       for (int i = 0; i < buf.limit(); ++i) {
4               byte b = buf.get(i);
5               // do something with b
6       }
```

Figure 4.1: Reading a buffer via absolute indexing

The value *position* is used for relative indexing. It is the index of the element that is read or written next, and is incremented after a data transfer operation. If *position* is greater or equal to *limit*, the methods `get` and `put` throw a `BufferUnderflowException` or a `BufferOverflowException`, respectively. Figure 4.2 shows an example of reading the contents of a buffer using relative indexing. The method `rewind()` sets the *position* to zero.

```
1       for(buf.rewind(); buf.position() < buf.limit();) {
2               byte b = buf.get();
3               // do something with b
4       }
```

Figure 4.2: Reading a buffer via relative indexing

4.1.2 Writing a `ByteBuffer`

Two methods come in handy when writing data into a buffer: `clear()` and `flip()`. The first of the two sets *position* to zero and *limit* to *capacity*. After its invocation, we are ready to write into the buffer using relative indexing, *i.e.*, the method `put(byte b)`. At most *capacity* many bytes can be put into the buffer in this way. When done with putting data into the buffer, method `flip()` is used to make the buffer ready to be read: This method sets *limit* to the current *position*, and then sets *position* to zero.

Figure 4.3 exemplifies this data writing procedure in the form of Java code.

```
1       ByteBuffer buf = ... // create new buffer
2
3       buf.clear();
4       for(int i = 0; i < dataSize; ++i) {
5               buf.put(0x2A);
6       }
7       buf.flip();
```

Figure 4.3: Writing a buffer

4.1.3 Creating a `ByteBuffer`

The class `java.nio.ByteBuffer` is abstract. Hence its constructor cannot be invoked. So how to create an object of type `ByteBuffer`?

The class `ByteBuffer` has two (non-abstract) static methods `ByteBuffer allocate(int capacity)` and `ByteBuffer wrap(byte[] array)`, which can be used for this purpose.

`ByteBuffer.allocate(42)` returns a `ByteBuffer` with *capacity* equal to 42. All of its entries are initialized to zero. For an array `byte[] array`, the invocation of method `ByteBuffer.wrap(array)` returns a `ByteBuffer` with *capacity* equal to `array.length` and its k^{th} entry equal to `array[k]`.

4.1.4 Bulk Data Transfer

Rather than calling the methods `get()` and `put(byte b)` for every single byte we want to read or write, it is possible to utilize the bulk data transfer methods, which operate on byte arrays. These methods are `ByteBuffer get(byte[] dst)` and `ByteBuffer put(byte[] src)`. Analogous methods exist with another `ByteBuffer` instead of a byte array.

Figure 4.4 shows two methods of reading bytes from a buffer into an array. The first method uses bulk data transfer, the second a `for`-loop.

```
1           ByteBuffer buf = ... // initialize buffer
2           byte[] array = new byte[BULK_SIZE];
3
4           // method 1
5           buf.rewind();
6           buf.get(array);
7
8           // method 2
9           buf.rewind();
10          for(int i = 0; i < BULK_SIZE; ++i)
11                  array[i] = buf.get();
```

Figure 4.4: Reading bulk data from a buffer

4.2 Basic Channel Usage

The base interface for all Channels is `java.nio.channels.Channel`. Like the abstract base class `Buffer`, it does not specify any methods for data transfer. These methods are specified in subinterfaces. Nevertheless, interface `Channel` does define the methods `isOpen()` and `close()`. After use, a channel should be closed. It is not possible to read from or write to a closed channel.

The interface `ReadableByteChannel` defines the method `int read(ByteBuffer dst)`, which reads bytes into buffer `dst`. The effect of this method on the buffer `dst` is the same as n repeated calls to `dst.put(byte b)`, where n is the number of bytes available in the channel. Whether this includes bytes not immediately available is dependent on whether the channel is declared to be blocking or not. The return value of the `read`-method is n, the number of bytes written into the buffer; or -1 if end-of-stream is reached. Figure 4.5 illustrates the use of the `read`-method.

```
1           ReadableByteChannel chan = ... // initialize channel
2           ByteBuffer buf = ByteBuffer.allocate(BUF_SIZE);
3
4           int n = chan.read(buf);
5           if(n != -1)
6                   System.out.println("There were "+n+" bytes read.");
7           else
8                   System.out.println("End-of-stream reached.");
9
10          chan.close();   // if no longer used
```

Figure 4.5: Reading from a channel

The interface `WritableByteChannel` analogously defines a method for writing to a channel, namely `int write(ByteBuffer src)`.

4.3 Blocking vs. Non-Blocking Operations

Blocking operations can have significant drawbacks. Say, we want to listen on a number of sockets and output incoming characters for the user on the screen. If there are no strict fairness guarantees on the relative rates of incoming packets on the sockets, with a blocking `read()` method, one thread for every socket is needed; each of which listens for incoming packets on a single socket. For if there would not be a seperate thread for every stream, it would be possible for one slow stream to block execution of the whole program. Figure 4.6 shows an implementation of these threads.

```
1  class SocketListener implements Runnable {
2          private final Socket socket;
3
4          public SocketListener(Socket socket) throws NullPointerException {
5                  if (socket == null)
6                          throw new NullPointerException();
7                  this.socket = socket;
8          }
9
10         public void run() {
11                 try {
12                         InputStream in = socket.getInputStream();
13
14                         int c = in.read();
15                         while(c != -1) {
16                                 System.out.print((char) c);
17                                 c = in.read();
18                         }
19                 } catch (IOException e) {
20                         // handle exception
21                 }
22         }
23 }
```

Figure 4.6: Blocking operations: Creating a thread for every socket

One advantage of this first approach is the clear structural representation of partitioning the problem into one sub-problem per socket. Also, because scheduling of threads is done by the JVM, there is no need to explicitly take care of the case that one of the threads, for any reason, is blocked.

If non-blocking operations are used, as they can be provided by channels, it is possible to do the same task inside a single main loop: Repeatedly cycle through all sockets and call their read-method, which immediately returns, and check if there is new data in the channel. Figure 4.7 contains a program skeleton for non-blockingly reading from sockets inside a single main loop.

In the second approach, the problem is also partitioned into one sub-problem per socket

```
1       SocketChannel[] sockets = ...  // open non-blocking sockets
2
3       for(;;) {
4           for(int i = 0; i < sockets.length; ++i) {
5               // read sockets[i]
6               // and print if data ready
7           }
8       }
```

Figure 4.7: Non-blocking operations: Having a single main loop

by having each loop iteration dealing with a single socket. This partioning, however, is not an explicit structural-syntactic one. Instead, there is a single thread and thus the JVM creates no overhead for managing multiple threads.

Also, scheduling is explicit. The order in which the sockets are checked for new data is explicitly stated. Doing explicit scheduling highlights another advantage of non-blocking operations: It is possible to write (approximately) *timing-predictable* programs, because the duration of each non-blocking operation can be bounded effectively.

4.4 Selectors

The `Selector` class in package `java.nio.channels` provides a way to implement the main loop of Figure 4.7 in a more efficient manner: With a `Selector` it is possible, instead of always looping over *all* sockets, to only loop over sockets that are presently ready for an operation (e.g., a read or write operation). Channels can be registered with a `Selector` for a given set of operations. The possible operations are `OP_ACCEPT`, `OP_CONNECT`, `OP_READ`, and `OP_WRITE` which are constant integers in class `SelectionKey`. (An object of type `SelectionKey` represents the registration of a specific channel with a specific selector.)

For this to work, one first creates an object `Selector sel` by calling the static method `Selector.open()` which returns a newly created `Selector` object. Then, it is possbile to register multiple channels of type `SelectableChannel` with `Selector sel`. This is done by calling the method `register(Selector sel, int ops)` on the channel. Here, the integer `ops` is a logical OR of one or more of the aforementioned operations, e.g., `OP_READ | OR_WRITE`. After registering channels with selector `sel`, its method `sel.select()` can be called. This method updates the set `sel.selectedKeys()` of `SelectionKeys` for channels ready for one of the registered operations. The call to `sel.select()` is blocking until at least one channel is ready for an operation. There also exists exists the non-blocking variant `sel.selectNow()`.

A `SelectionKey key` contains a reference `key.channel()` to the channel that was selected. The integer `key.readyOps()` states for which operations the channel is ready.

This information on the operations that the channel is ready for can also be accessed via the methods `key.isAcceptable()`, `key.isConnectable()`, `key.isReadable()`, and `key.isWritable()`. For example, `key.isReadable()` is equal to the Boolean value `key.readyOps() & SelectionKey.OP_READ`.

To end a registration of a channel with a selector, wither the channel or the selector can be closed, or the `SelectionKey`'s `cancel()` method can be called.

To attach and retrieve application specific data to a registered channel, `SelectionKey key` provides the two methods `key.attach(Object ob)` and `key.attachment()`.

4.5 A High-Performance Server

This section shows an example of using channels with non-blocking operations for implementing a network server. The program has two threads: one for accepting new incoming connections and one for handling the existing ones. In particular, it does not create a new thread for every connection.

The server's functionality is the following: The server waits for a client's message and returns the message, after adding a per-connection sequence number to it, back to the client. After returning the 4^{th} message to the client, the server closes the connection. Figure 4.8 shows the logical implementation of this functionality. Figure 4.9 shows an example client session with the server.

```
1  private class ConnectionState {
2          private String msg = null;
3          private int seqNum = 0;
4
5          void receive(String msg) throws NullPointerException {
6                  if(msg == null)
7                          throw new NullPointerException();
8                  else
9                          this.msg = msg;
10         }
11
12         String send() {
13                 ++seqNum;
14                 return "Message number " + seqNum +  " from you: " + msg;
15         }
16
17         boolean toBeClosed() {
18                 return (seqNum >= 4);
19         }
20 }
```

Figure 4.8: Class `ConnectionState` to be instantiated for every connection

```
tnowak@oahu:~$ telnet localhost 4242
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hi
Message number 1 from you: Hi
Huhu
Message number 2 from you: Huhu
Haha
Message number 3 from you: Haha
Hello Goodbye
Message number 4 from you: Hello Goodbye
Connection closed by foreign host.
tnowak@oahu:~$
```

Figure 4.9: A client session

Figure 4.10 shows the thread that accepts incoming connections on `ServerSocketChannel serverChan` and registers them with `Selector sel` for a read operation (the application logic starts with waiting for a client's message). Its main loop contains a blocking call to `serverChan.accept()`, after which it registers the new `SocketChannel` with the selector and attaches a new object of type `ConnectionState` to the registration (i.e., the correpsonding `SelectionKey`).

The main loop of the thread handling the individual connections is contained in Figure 4.11. It starts with a selection call to the selector, after which all selected channels are handled, depending on whether its next operation (the operation it was registered for and for which it is now ready) is a read or a write operation. The handling of these two cases is in Figures 4.12 and 4.13. The variables `charBuffer`, `enc`, and `dec` are for correctly encoding and decoding the strings for class `ConnectionState` in the `ByteBuffer msgBuffer`, which is ultimately used to read from or write to the channels. The method `key.interestOps(int ops)` changes the operations that channel `key.channel()` is registered for. It is used in the example to alternately read from and write to the channel.

The rest of the (main) class `Server` is depicted in Figure 4.14.

```
1  private class Acceptor implements Runnable {
2          public Acceptor() throws IOException {
3                  sel = Selector.open();
4          }
5
6          public void run() {
7                  SocketChannel chan;
8                  SelectionKey key;
9
10                 for(;;) {
11                         try {
12                                 chan = serverChan.accept();
13                                 chan.configureBlocking(false);
14                                 key = chan.register(sel, SelectionKey.OP_READ);
15                                 key.attach(new ConnectionState());
16                         } catch(IOException e) {
17                                 e.printStackTrace();
18                         }
19                 }
20         }
21 }
```

Figure 4.10: Thread for accepting new connections

```
1  public void run() {
2          ByteBuffer msgBuffer = ByteBuffer.allocateDirect(MAX_MSG_SIZE);
3          CharBuffer charBuffer = CharBuffer.allocate(MAX_MSG_SIZE);
4          CharsetEncoder enc = Charset.forName(CHARSET).newEncoder();
5          CharsetDecoder dec = Charset.forName(CHARSET).newDecoder();
6
7          for(;;) {
8                  try {
9                          sel.selectNow();
10                 } catch(IOException e) {
11                         e.printStackTrace();
12                         System.exit(1);
13                 }
14
15                 for(SelectionKey key : sel.selectedKeys()) {
16                         sel.selectedKeys().remove(key);
17                         SocketChannel chan = (SocketChannel) key.channel();
18                         ConnectionState state = (ConnectionState) key.attachment();
19
20                         if (key.isValid() && key.isReadable()) {
21                                 // do read
22                         }
23
24                         if (key.isValid() && key.isWritable()) {
25                                 // do write
26                         }
27                 }
28         }
29 }
```

Figure 4.11: Main server loop

```
1         if (key.isValid() && key.isReadable()) {
2                 msgBuffer.clear();
3                 try {
4                         if( chan.read(msgBuffer) == -1)
5                         {
6                                 chan.close();
7                                 continue;
8                         }
9                 } catch(IOException e) {
10                        e.printStackTrace();
11                }
12                msgBuffer.flip();
13                charBuffer.clear();
14                dec.decode(msgBuffer, charBuffer, true);
15                charBuffer.flip();
16                state.receive(charBuffer.toString());
17
18                key.interestOps(SelectionKey.OP_WRITE);
19        }
```

Figure 4.12: Receiving a message

```
1         if (key.isValid() && key.isWritable()) {
2                 charBuffer.clear();
3                 charBuffer.put(state.send());
4                 charBuffer.flip();
5                 msgBuffer.clear();
6                 enc.encode(charBuffer, msgBuffer, true);
7                 msgBuffer.flip();
8                 try {
9                         if( chan.write(msgBuffer) == -1) {
10                                chan.close();
11                                continue;
12                        }
13                } catch(IOException e) {
14                        e.printStackTrace();
15                }
16
17                key.interestOps(SelectionKey.OP_READ);
18
19                if(state.toBeClosed()) {
20                        try {
21                                chan.close();
22                        } catch(IOException e) {
23                                e.printStackTrace();
24                        }
25                }
26        }
```

Figure 4.13: Sending a message

```
1  public class Server implements Runnable {
2          private static final int PORT = 4242;
3          private static final int MAX_MSG_SIZE = 1024;
4          private static final String CHARSET = "UTF-8";
5          private ServerSocketChannel serverChan;
6          private Selector sel;
7
8          public Server(int port) throws IOException {
9                  serverChan = ServerSocketChannel.open();
10                 serverChan.configureBlocking(true);
11                 serverChan.socket().bind(new InetSocketAddress(port));
12         }
13
14         ...
15
16         public static void main(String[] args) {
17                 try {
18                         Server server = new Server(PORT);
19                         Thread serverThread = new Thread(server);
20                         Thread acceptThread = new Thread(server.new Acceptor());
21                         serverThread.start();
22                         acceptThread.start();
23                 } catch (IOException e) {
24                         e.printStackTrace();
25                         System.exit(1);
26                 }
27         }
28 }
```

Figure 4.14: Class `Server`

Contents