23 Error Handling

- Exit program (`System.exit()`)
  ◆ usually a bad idea
- Output an error message
  ◆ does not help to recover from the error
- Special error return
  ◆ Constructors do not have a return value
  ◆ What if method uses the full range of the return type?
- Call a user defined error handler
  ◆ awkward
  ◆ What must this method do?
- Exceptions!

23.1 What happens when a method is called?

```java
class Customer {
    void createAccount(Bank bank) {
        Account account = new Account();
        bank.newAccount(account, 5);
    }
}
class Bank {
    void newAccount(Account a, int i) {
        int counter = 0;
        ...
    }
}
class Main {
    public static void main(String args[]) {
        Customer c = new Customer();
        c.createAccount(new Bank());
    }
}
```

```
Stack frame of:
class Main:
Stack frame of:
class Bank:
Stack frame of:
class Customer:
```

23.1 What is Exception Handling?

- Transfer control from error origin to error handler

```
A.m() 
B.n() 
C.p() 
```

- Responsibilities:
  ◆ Code author can detect the error but doesn’t know how to handle it.
  ◆ Code user can handle the error but cannot detect it.
23.1 What happens when a method is called?

```java
class Customer {
    void createAccount(Bank bank) {
        Account account = new Account();
        bank.newAccount(account, 5);
    }
}
class Bank {
    void newAccount(Account a, int i) {
        int counter = 0;
        // ... 
    }
}
class Main {
    public static void main(String[] args) {
        Customer c = new Customer();
        c.createAccount(new Bank());
    }
}
```

Stack frame of `main`
- **args**: parameter variable
- **c**: local variable

Stack frame of `createAccount`
- **account**: parameter variable
- **bank**: local variable

Stack frame of `newAccount`
- **account**: local variable
- **counter**: parameter variable
- **i**: parameter variable

Stack
- **newAccount**: method call
- **createAccount**: method call
- **main**: method call
23.2 Try, Throw, and Catch

```java
try {
    ...
    if (...) throw new MyException();
    ...
} catch (MyException e) {
    // exception handler
    ...
}
```

- use throw to throw an exception
- catch block must immediately follow try block
- there can be more than one catch block
  - catch blocks are matched in program order
- a method may not catch all exceptions
  - uncaught exceptions are automatically thrown up the stack

23.3 Finally

```java
try {
    ...
} catch(...) {
    // error handling
    ...
} finally {
    // release resources
    ...
}
```

- the finally block is executed if the try block was entered
  - can be used to clean up in case of (un-)caught exceptions
- a `finally` block can also be used without `catch`

23.4 throws

```java
class Test {
    void m() throws MyException {
        ...
        if (...) throw new MyException();
        ...
    }
}
```

- Exceptions must be declared in method header

23.5 Error Classes

- All exceptions are derived from `Throwable`
- Exceptions that can be expected nearly everywhere:
  - `Error`: Linker errors, errors in the format of class files, out of memory, ...
  - `RuntimeException`: array index, null pointer, illegal cast, arithmetic, ...
- Application-program exceptions are derived from `java.lang.Exception`

![Error Classes Diagram]

- needs not to be declared or caught
### 23.6 Exceptions and Inheritance: Catching

- Catching exception subclasses with multiple catch blocks
- Notice: Superclasses match all subclasses, catch superclasses at last

```java
class MathException {}
class ZeroDivideException extends MathException {}
class InvalidArgException extends MathException {}
try {
  ...
  catch (ZeroDivideException e) {
    ...
  }
  catch (InvalidArgException e) {
    ...
  }
  catch (MathException e) {
    ...
}
```

### 23.7 Exceptions and Inheritance: Throwing

- Can overriding method throw other exceptions than the original method?
- Principle:
  - Subclasses can be used wherever a superclass is expected.
  - Subclasses are "better" than superclasses.
- This means:
  - Subclass must not throw more exceptions than superclass.
  - Subclass may throw subclasses of the superclass-thrown exceptions.
  - Subclass must not throw superclasses of the exceptions.

### Example

```java
class TestException extends Exception {
  public TestException(String s) { super(s); }
}

class Test {
  public void hello() throws TestException {
    if (...) throw new TestException("...an error description...");
  }

  public void testIt() {
    try {
      hello();
    } catch (TestException t) {
      System.out.println("Exception raised:" + t.getMessage());
    } finally {
      // clean up
    }
  }
}
```

### Example

```java
class E1 extends Exception {}
class E2 extends Exception {}
class E3 extends E2 {}
class A {
  void m() throws E2 {}
}
class B extends A {
  void m() throws ??? {}
}
```

```java
??? =
```
### 23.8 Beispiel

```java
class E1 extends Exception {}
class E2 extends Exception {}
class E3 extends E2 {}
class A {
    void m() throws E2 {}
}
class B extends A {
    void m() throws ??? {}  
}
```

<table>
<thead>
<tr>
<th>Wrong:</th>
<th>Correct:</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1 Exception ...</td>
<td>E2 E3 none</td>
</tr>
</tbody>
</table>

### 23.9 Zusammenfassung

- Werfen einer Ausnahme: `throw new MyException("...");`
- Block, der Ausnahmen werfen kann: `try { ... }`
- Ausnahmebehandlung: `try {
                ... throw new MyException("...");
                }
     catch(MyException e) {
                ...
              }`
- Zusätzlich: `finally` Block
- Ausnahmen müssen Subklassen von `Throwable` sein.
- Anwendungsaufrufe müssen Subklassen von `Exception` sein.
- Ausnahmen müssen im Methodenkopf mit `throws` deklariert werden.

### 24 Datei-IO-System

- **Basiskonzept**: Streams
- Byte streams: (InputStream/OutputStream)
- Zeichenströme: (Reader/Writer)

### 24.1 Byte Streams

- **Grundkonzept**: Ströme
  - Bitströme (InputStream/OutputStream)
  - Zeichenströme (Reader/Writer)

![Diagramm zu Byte Streams]
24.2 Stream Specializations

- Where do the data come from / go to?

FileInputStream
File
FileOutputStream
ByteArrayInputStream
byte[]
ByteArrayOutputStream
SocketOutputStream
SocketInputStream
Internet
SocketOutputStream
SocketInputStream
Internet
?
### 24.3 Input Streams Class Diagram

- **InputStream**
  - ByteArrayInputStream
  - FileInputStream
  - PipedInputStream
  - FilterInputStream
  - SequenceInputStream
  - DataInputStream
- **BufferedInputStream**
- **PushbackInputStream**
- **DataInput**
- **ObjectInput**

### 24.4 FileInputStream, FileOutputStream

- Read from a file

```java
import java.io.*;

public class Test {
    public static void main (String argv[]) throws IOException {
        FileInputStream f = new FileInputStream("/tmp/test");
        byte buf[] = new byte[4];
        f.read(buf);
    }
}
```

### 24.5 Combining Streams

- Create comfortable streams from simple streams
- The comfortable stream wraps the simple stream
- ➜ Decorator Design-Pattern

### 24.6 Combining Streams

- **User**
- **DataSource**
- **ByteStream**
- **InputStream**
24.7 Combining Streams

- User reads data from an InputStream, which can then be read by a DataInputStream.

24.8 DataInputStream

- InputStream is rather uncomfortable.
- DataInputStream is used to read the binary representation of data (int, float,...).
- It can be created from every InputStream.

```java
InputStream in = new FileInputStream("/tmp/test");
DataInputStream dataIn = new DataInputStream(in);
float f = dataIn.readFloat();
```

- `readLine()` can be used to read whole lines.

```java
for(;;) {
    String s = dataIn.readLine();
    System.out.println(s);
}
```

24.9 Reader/Writer

- Character streams for input/output (`Reader`, `Writer`).
- Character streams are used to read and write characters containing Unicode characters (16 bit).

24.10 Reader

- Important methods:

  ```java
  int read()
  Read one character and return it as int.
  int read(char buf[])
  Read characters into buffer. Return number of read characters or -1 in case of error.
  int read(char buf[], int offset, int len)
  Read len characters in buffer buf starting at offset.
  long skip(long n)
  Skip n characters.
  void close()
  Closes the stream.
  ```
24.10.1 FileReader

- Used to read from file
- Constructors:
  - FileReader(String fileName)
  - FileReader(File file)
  - FileReader(FileDescriptor fd)
- No additional methods (only inherited from InputStreamReader)
- What is an InputStreamReader?

24.11 Byte Streams and Character Streams

- Convert byte streams to character streams using an encoding
- some encodings: "Basic Latin", "Greek", "Arabic", "Gurmukhi"

24.12 Buffered IO

- Reading/writing single characters can be expensive.
- Converting encodings can be expensive.
- Use BufferedReader, BufferedWriter if possible.
- BufferedReader can be created from every other Reader.
- Important method of BufferedWriter:
  - void flush(): Empties the buffer - writes buffer to unbuffered writer.

24.13 Buffered IO

- BufferedReader can read whole lines: String readLine()

```java
BufferedReader in = new BufferedReader(new FileReader("test.txt"));
String line = in.readLine();

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line = in.readLine();
```
24.14 PrintWriter

- Can be created from every OutputStream or Writer
- println(String s): write string and end-of-line character(s)
- Example: Read file and print it to standard output

```java
import java.io.*;

public class CopyStream {
    public static void main(String a[]) throws Exception {
        BufferedReader in = new BufferedReader(
            new FileReader("test.txt"));
        PrintWriter out = new PrintWriter(System.out);
        for(String line; (line = in.readLine())!=null;) {
            out.println(line);
        }
        out.close();
    }
}
```

24.15 FileWriter

- used to write characters to a file
- invoke close() after you finished writing!