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 is Exception Handling?

- Transfer control from error origin to error handler

![Diagram showing A.m() function with B.n() nested within it, and C.p() as a separate function.]

- 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`
- `c`

Parameter variable

Local variable
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 `createAccount`
- Parameter variable (bank)
- Local variable (account)

Stack frame of `main`
- Parameter variable (args)
- Local variable (c)

Stack

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.
23.1 What happens when a method is called?

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 createAccount

Stack frame of newAccount

Stack frame of main

Stack

| parameter variable (a) |
| parameter variable (bank) |
| local variable (account) |
| parameter variable (counter) |
| local variable |
| local variable |
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 `createAccount`
- `bank`
- `account`

Stack frame of `main`
- `args`
- `c`
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());
    }
}
```
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
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

- the finally block is executed if the try block was entered
  - can be used to clean up in case of (un-)caught exceptions

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

- a `finally` block can also be used without `catch`

```java
try {
    ...
    if (...)
    return;
    ...
} finally {
    ... } 
```
23.4 throws

- Exceptions must be declared in method header

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

Diagram:
- `Throwable` is the root of the exception hierarchy.
- `Exception` is a superclass of `Error` and `RuntimeException`.
- `Error` and `RuntimeException` are specific types of exceptions.
- `getMessage()` is a method inherited from `Throwable`, allowing retrieval of exception messages.

Note: `Error` and `RuntimeException` do not need to be declared or caught.
Catching exception subclasses with multiple catch blocks

Notice: Superclasses match all subclasses, catch superclasses at last

class MathException {}  
class ZeroDivideException extends MathException {}  
class InvalidArgException extends MathException {}

try {
    ...
} catch(ZeroDivideException e) {
    ...
} catch(InvalidArgException e) {
    ...
} catch(MathException e) {
    ...
}
Example

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

public 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
        }
    }
}
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.
23.8 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 ??? {}
}
```

`??? =`
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 ??? {}
}
```

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

- Throwing an exception: `throw new MyException("..."`;

- Block that may throw exceptions: `try { ... }`

- Handling exceptions:
  ```java
  try {
  ... throw new MyException("...")); ... 
  } catch(MyException e) { ... }
  ```

- Additionally: `finally` block

- Exceptions must be subclasses of `Throwable`.

- Application exceptions should be subclasses of `Exception`.

- Exceptions must be declared in the method header with `throws`
Basic concept: streams

- Byte streams (InputStream/OutputStream)
- Charater streams (Reader/Writer)
24.1 Byte Streams

![Diagram of Byte Streams]

User

read()  write()

Data Stream

InputStream

OutputStream

Data Sink

Data Source

Byte-Stream

Byte-Stream
Stream Specializations

Where do the data come from / go to?

- File
  - FileInputStream
  - FileOutputStream
24.2 Stream Specializations

Where do the data come from / go to?

- FileInputStream
- FileOutputStream
- ByteArrayInputStream
- ByteArrayOutputStream

File

byte[]
24.2 Stream Specializations

Where do the data come from / go to?

- FileInputStream
- FileOutputStream
- ByteArrayInputStream
- ByteArrayOutputStream
- SocketInputStream
- SocketOutputStream
- ByteBuffer
- SocketOutputStream
- Internet
24.2 Stream Specializations

- Where do the data come from / go to?

- File
  - FileInputStream
  - FileOutputStream

- ByteArrayInputStream
  - byte[]
  - ByteArrayOutputStream

- SocketInputStream
  - Internet

- SocketOutputStream
24.3 Input Streams Class Diagram

InputStream

ByteArrayInputStream  FileInputStream  PipedInputStream

FilterInputStream  SequenceInputStream

BufferedInputStream  PushbackInputStream  DataInputStream

DataInput

ObjectInput

ObjectInputStream
### 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

read()

DataService

Byte-Stream

InputStream
Combining Streams

```
Data Source -> Byte-Stream -> InputStream -> DataInputStream
```

```
User
```

```
readInt()
```
24.8 DataInputStream

- InputStream rather uncomfortable
- DataInputStream used to read *binary representation* of data (int, float,...)
- 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 contain Unicode characters (16 bit)
### 24.10 Reader

**important methods:**

* `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!