

## Abend 3

### Notenliste fertigstellen, Ausnahmebehandlung

#### Ziel, Inhalt

- Heute werden wir lernen die TsuZeichnen-Komponente zu verwenden um eine einfache Notengrafik darzustellen
- Wir sind ab heute in der Lage die C++-Ausnahmebehandlung zu verstehen und anzuwenden

Abend 3 Notenliste fertigstellen, Ausnahmebehandlung	1
Ziel, Inhalt	1
Grafische Ausgabe für die Notenliste	2
Verwenden von TsuZeichnen	2
Übung Grafische Ausgabe	2
Ausnahmebehandlung	3
Traditioneller Ansatz	3
try, catch, throw	4
Übung Klasse Zahl	7

## Grafische Ausgabe für die Notenliste

### Verwenden von TsuZeichnen

Wie bereits erwähnt, stehen einige Klassen und eine ausführbare Datei zur Verfügung um grafische Ausgaben zu tätigen. Lese die Anleitung in diesem Dokument nach:

[www.devmentor.ch/teaching/additional/01/TsuZeichnen/UebungZeichnen.pdf](http://www.devmentor.ch/teaching/additional/01/TsuZeichnen/UebungZeichnen.pdf).

Falls du dich für den Server interessierst, findest du hier die nötigen (aber kanppen) Informationen:

<http://www.devmentor.ch/teaching/additional/01/Semester3/index.html>.

### Übung Grafische Ausgabe

Eine Möglichkeit besteht darin für jede Notenliste eine eigene Zeichenfläche zu verwenden. Dies scheint auch der Vorgabe zu entsprechen (siehe [Abend 1](#)).

Sauber scheint mir hier ein Objekt der Klasse ZeichenFlaeche als Datenelement der Klasse Notenliste anzulegen.

Du kannst sobald der Benutzer den Namen des Faches angegeben hat, die ZeichenFlaeche dynamisch erzeugen und ein Text-Objekt erzeugen um den Namen des Faches anzuzeigen. Falls Du im Konstruktor von Notenliste als Konstruktor-Argument den Namen des Faches hast, kannst Du direkt im Konstruktor die ZeichenFlaeche initialisieren. Beachte, dass die ZeichenFlaeche nur einen Konstruktor mit Parametern (für die Grösse) hat. Das heisst du musst das ZeichenFlaeche-Objekt entweder dynamisch erzeugen oder in der Initialisierungsliste des Konstruktors initialisieren. Das Text-Objekt kannst du einfach erzeugen, den Inhalt definieren und dann mit ZeichenFlaeche::textHinzufuegen darstellen lassen.

Um eine Liniendarstellung zu erreichen musst du einerseits für jede Note (ausser der ersten) eine Linie erzeugen und andererseits die bereits bestehenden Linien anpassen, da sich jeweils die Anfangs- und Endpunkte ändern können. Die Linien-Objekte speicherst du selbstverständlich in einem Container!

Sobald die zweite Note eingegeben wurde kannst du damit beginnen Linien zur Grafik hinzuzufügen. Schreibe also eine Methode „addLine“ oder so. In dieser Methode wirst du aus Noten y-Werte für die Linien-Koordinaten berechnen müssen. Du kannst diese Rechnerei in einer privaten Methode von Notenliste erledigen. Ein völlig anderer Ansatz wäre es die Linie selber so intelligent zu machen, dass sie aus einem Noten-Objekt ein y-Koordinate berechnen kann. Hierfür müsstest Du von der Klasse Linie ableiten. Ich würde hier den ersten Ansatz weiterverfolgen... Die x-Koordinaten berechnest du ganz einfach aus der Breite der ZeichenFlaeche und der Anzahl Linien.

## Ausnahmebehandlung

### Traditioneller Ansatz

Es gibt den Ansatz, dass alle Methoden oder Funktionen, die irgendwie schiefgehen können als Rückgabewert einen Indikator haben, ob die Methode funktioniert hat. Das ist manchmal ein boole'scher Wert, oder kann sogar ein Fehlercode sein. Das führt manchmal zu schwer lesbarem Code, da ein grosser Teil des Codes Fehlerbehandlung ist. Hier eine fiktive File-Klasse:

```
class File
{
    public:
        bool open(const char* filename);
        bool read(unsigned char& data);
        bool write(const unsigned char& data);
        bool close();
};

int main()
{
    File einFile;
    if(einFile.open(„hallo.txt“);
    {
        unsigned char data = 0;
        if(einFile.read(data))
        {
            if(doSomething(data))
            {
                if(einFile.write(data))
                {
                    // everything ok
                }
                else
                {
                    writeFailed();
                }
            }
            else
            {
                doSomethingFailed();
            }
        }
        else
        {
            readFailed();
        }
    }
    if(einFile.close())
    {
        // ja und?
    }
    else
    {
        und jetzt?
    }
}
```

```
    }
    else
    {
        openFailed();
    }
    return 0;
}
```

Nicht besonders schön. Man kann hier einige Dinge voraussetzen. Nämlich, dass falls das Öffnen funktioniert, funktioniert das Lesen auch immer und die `doSomething`-Funktion sollte auch immer funktionieren, etc. Im allgemeinen schreibt man Methoden und Funktionen ja damit sie funktionieren! Wenn sie nicht funktionieren muss eine Ausnahmesituation vorliegen. Trotzdem ist unser Code kaum mehr lesbar und man versteht schwer, was das Programm überhaupt machen soll.

### try, catch, throw

Es gibt nun in C++ die Möglichkeit den Normalfall zu programmieren und Ausnahmen auch als solche zu behandeln. Dieses Programm versucht einen Programmteil ablaufen zu lassen:

```
int main()
{
    try
    {
        File einFile;
        einFile.open(„hallo.txt“);
        unsigned char data = 0;
        einFile.read(data);
        doSomething(data);
        einFile.write(data);
        einFile.close();
    }
    ....
    return 0;
}
```

Mit einem `try`-Block erzeugen wir einen Bereich in dem C++-Exception auftreten dürfen. Tritt eine Ausnahme ein, wird der `try`-Block verlassen, es wird aus dem Block herausgesprungen. Das besondere dabei ist, dass dabei alle bereits im Block erstellten Objekte korrekt gelöscht werden. Das heisst, es werden alle Destruktoren von erzeugten Objekten aufgerufen. Wir können mit diesem Wissen den Destruktor von der `File`-Klasse implementieren:

```
class File
{
public:
    ~File()
    {
        // always clean up!
        close();
    }
}
```

Der Code von oben wird so noch einfacher:

```
int main()
{
    try
    {
        File einFile;
        einFile.open(„hallo.txt“);
        unsigned char data = 0;
        einFile.read(data);
        doSomething(data);
        einFile.write(data);
    }
    ....
    return 0;
}
```

Tritt eine Ausnahme ein wird das File-Objekt zerstört und auf jeden Fall geschlossen (siehe Destruktor)!

Ausnahmen sind Objekte, die „geworfen“ werden. Eine Methode in der eine Ausnahme auftritt, wirft mit *throw* ein Objekt oder auch einfach ein Wert eines eingebauten Datentypen. Die Methode gibt keinen bool mehr zurück sondern wirft im Notfall irgend ein Objekt. Hier gleich als Beispiel die File-Klasse mit Ausnahmen. Als Fehlerobjekt wird hier ein string-Objekt geworfen. Beachte auch, wie die Methoden keinen bool mehr zurückgeben, oder die Methode read jetzt den gelesenen Wert als Rückgabewert hat:

```
class File
{
public:
    void open(const char* filename);
    unsigned char read() const;
    void write(const unsigned char& data);
    void close();
};

void File::open(const char* filename)
{
    // open file
    if(!isOpen())
    {
        string fehler(„File nicht offen!“);
        throw(fehler);
    }
}

unsigned char File::read() const
{
    if(!isOpen())
    {
        string fehler(„Datei nicht offen beim Lesen“);
        throw fehler;
    }
}
etc.
```

Dinge die geworfen werden, müssen auch gefangen werden. Fangen heisst *catch* auf Englisch. Bei *catch* müssen wir einfach noch angeben was gefangen werden muss. Hier das *main* von vorhin, das *string*-Objekte (mit Fehlertext) fängt und den Fehlertext in der Konsole ausgibt:

```
int main()
{
    try
    {
        File einFile;
        einFile.open(„hallo.txt“);
        unsigned char data = 0;
        einFile.read(data);
        doSomething(data);
        einFile.write(data);
    }
    catch(const string& error)
    {
        cout << „An error has occurred“ << endl;
        cout << error << endl;
    }
    ....
    return 0;
}
```

Beachte hierbei, dass der *catch*-Block direkt hinter den *try*-Block gehört. Wir fangen in diesem *catch* Block also alles, was vom Datentypen her einem *string*-Objekt entspricht.

Es können mehrere *catch*-Blöcke hintereinander stehen, denn es könnten verschiedene Objekte mit *throw* geworfen werden. Nehmen wir an es wird in der Methode *doSomething* ein *int* als Fehlercode geworfen und wir wollen diesen fangen. Das *main* sieht dafür so aus:

```
int main()
{
    try
    {
        File einFile;
        einFile.open(„hallo.txt“);
        unsigned char data = 0;
        einFile.read(data);
        doSomething(data);
        einFile.write(data);
    }
    catch(const string& error)
    {
        cout << „An error has occurred“ << endl;
        cout << error << endl;
    }
    catch(int errorCode)
    {
        cout << „An error has occurred“ << endl;
        cout << „error code : „ << errorCode << endl;
    }
}
```

Wird ein Fehler geworfen wird der erste passende catch-Handler gesucht.  
Mit catch(...) können alle Fehler gefangen werden:

```
int main()
{
    try
    {
        File einFile;
        einFile.open(„hallo.txt“);
        unsigned char data = 0;
        einFile.read(data);
        doSomething(data);
        einFile.write(data);
    }
    catch(const string& error)
    {
        cout << „An error has occurred“ << endl;
        cout << error << endl;
    }
    catch(int errorCode)
    {
        cout << „An error has occurred“ << endl;
        cout << „error code : „ << errorCode << endl;
    }
    catch(...)
    {
        cout << „An unspecified error has occurred“ << endl;
    }

    return 0;
}
```

## Übung Klasse Zahl

Die Klasse Zahl wird vorgestellt. Versuche zu verstehen wie diese funktioniert.

Ergänze die Klasse um Exceptions. Einige Methoden werfen bereits exceptions aus der STL (Standard Template Library). Du kannst eine eigene Klasse definieren und von der standard exception Klasse ableiten! Oder einfach diese verwenden!