

Einführung in die Programmierung

Vorlesungsteil 9

Zeichenketten, Ausnahmebehandlung und Arbeit mit Textdateien in Java

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik

Sommersemester 2016



Brandenburgische
Technische Universität
Cottbus - Senftenberg

Zeichenketten sind im Alltag präsent



Zeichenketten sind das in Datenbanken am häufigsten genutzte Datenformat.

- Auch scheinbar numerische Daten wie *Telefonnummern* (0355) 69-0 oder *Hausnummern* 42a
- Zeichenketten widerspiegeln die *menschliche Schriftsprache*, in der traditionell Daten archiviert wurden und werden

Zeichenketten sind vielseitig

$$\sqrt{x^2 + y^2} \cdot \begin{pmatrix} a \\ b \end{pmatrix}$$

```
$\sqrt{x^{2} + y^{2}} \cdot \left(a \atop b\right)$
```

Zeichenketten bieten eine universelle Datenstruktur.

- Beliebige Datenstrukturen wie *Zahlen*, *Bäume*, *Graphen* oder *Formeln* lassen sich verlustfrei und eindeutig durch wohldefinierte Abfolgen von Symbolen (Zeichen) darstellen
- Theorie *Formaler Sprachen* liefert algorithmische Werkzeuge zur Erzeugung und Analyse von Zeichenketten (z.B. Compiler einer Programmiersprache analysiert Quelltext und erzeugt Bytecode oder Maschinencode)

Dateien speichern Daten persistent



Eine Datei kann eine Zeichenkette langfristig computerlesbar archivieren.

- *Haltbarkeit* von Speichermedien für Dateien variiert je nach Nutzung und ist stets begrenzt
- **USB-Stick** bis zu ca. **5** Jahren
- **Festplatte** bis zu ca. **10** Jahren
- **CD / DVD** mindestens **30** Jahre (Prognose)
- **Holografische Speicher** > **100** Jahre (Prognose)

Vorhersehbare Fehlersituationen als Programmausnahmen

Manche *Fehler* kann man als Programmierer nicht vermeiden:

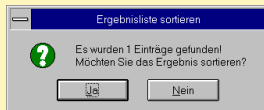
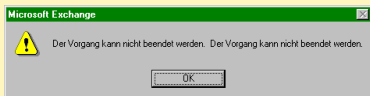
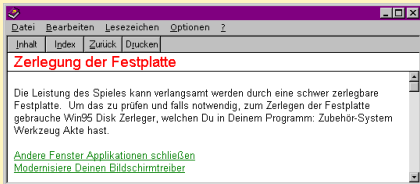
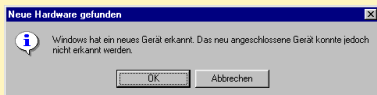
- Datei kann nicht angelegt werden, weil Datenträger (fast) voll
- Datei kann nicht geöffnet werden, weil sie nicht vorhanden ist
- Fehler beim Einlesen von Daten (z.B. aus einer Datei)
- Objekt kann nicht angelegt werden, weil nicht genug Speicher frei ist
- Netzverbindung nicht herstellbar
- Peripheriegerät nicht ansprechbar

Solche und ähnliche Situationen als *Ausnahmen* (*Exceptions*) abfangen und behandeln



Lustige Fehlermeldungen

Programmausnahmen ziehen meist Fehlermeldungen nach sich ...



www.seitenreport.de

Vorlesung Einführung in die Programmierung mit Java

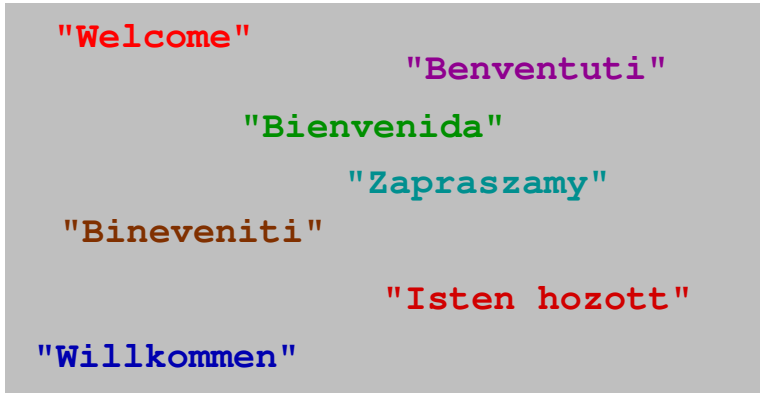
- 1. Einführung und erste Schritte**
.. Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
- 2. Elementare Datentypen, Variablen, Arithmetik, Typecast**
Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
- 3. Imperative Kontrollstrukturen**
... Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren
- 4. Methoden selbst programmieren**
.... Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe
- 5. Rekursion**
.... selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
- 6. Objektorientiert programmieren**
..... Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie
- 7. Felder und Graphen**
.... effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken
- 8. Sortieren**
..... klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
- 9. Zeichenketten, Dateiarbeit, Ausnahmen**
... Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln
- 10. Dynamische Datenstruktur „Lineare Liste“**
..... unsere selbstprogrammierte kleine Datenbank
- 11. Ausblick und weiterführende Konzepte**

Die 51 Schlüsselwörter von Java

Heute lernen wir davon kennen ...

abstract	double	long	static
boolean	else	native	super
break	extends	new	switch
byte	final	null	synchronized
case	finally	operator	this
cast	float	outer	throw
catch	for	package	throws
char	if	private	transient
class	implements	protected	try
const	import	public	var
continue	instanceof	rest	void
default	int	return	while
do	interface	short	

Zeichenketten (Strings)



Eine Zeichenkette ist eine endliche Abfolge von Symbolen (aus der Unicode-Tabelle).

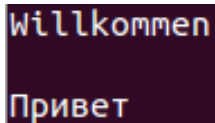
Zeichenkette in Java

Java stellt im Standardsprachumfang den *Typ* bzw. die *Klasse* **String** zur Verfügung. Ein String-Objekt ist eine konkrete Zeichenkette, deren Handling ähnlich wie Variablenwerte elementarer Datentypen erfolgt (z.B. *Literale zuweisen*) und ergänzt wird durch verfügbare Methoden (z.B. **length()**) wie bei Klassen üblich.

```
public class Zeichenketten {
    public static void main(String[] args) {

        String s1 = "Willkommen";           // Literal
        String s2 = "";                     // leere Zeichenkette
        String s3 = "\u041F\u0440\u0438\u0432\u0435\u0442"; // Unicode-Werte

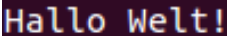
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```



Eine Zeichenkette darf beliebig, aber endlich lang sein. Literale werden beidseitig durch `"` begrenzt.

Zeichenketten formatiert am Monitor ausgeben

```
public class Zeichenketten2 {  
    public static void main(String[] args) {  
  
        String s1 = "Hallo";  
        String s2 = "Welt!";  
  
        System.out.printf("%s %s\n", s1, s2);  
    }  
}
```



Im **printf**-Formatstring markiert der Platzhalter **%s** eine Zeichenkette.

Zeichenketten formatiert am Monitor ausgeben

```
public class Zeichenketten3 {
    public static void main(String[] args) {

        String[] w = {"Montag", "Dienstag", "Mittwoch",
                     "Donnerstag", "Freitag", "Samstag", "Sonntag"};

        String[] a = {"Anreise", "Ausschlafen", "Meeting",
                     "Museumstour", "Strand", "Party", "Abreise"};

        System.out.printf("+-----+-----+\n");
        for (int i = 0; i < 7; i++)
        {
            System.out.printf("|%12s|%18s|\n", w[i], a[i]);
        }
        System.out.printf("+-----+-----+\n");
    }
}
```

```
+-----+-----+
| Montag |      Anreise |
| Dienstag |   Ausschlafen |
| Mittwoch |      Meeting |
| Donnerstag | Museumstour |
| Freitag |      Strand |
| Samstag |      Party |
| Sonntag |     Abreise |
+-----+-----+
```

Rechtsbündig ausgeben: z.B. **%12s** stellt der Zeichenkette in Ausgabe Leerzeichen voran, so dass insgesamt 12 Zeichen erscheinen

Zeichenkette von Tastatur eingeben per Scanner

```
import java.util.Scanner;

public class Zeichenketten4 {
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Eingabe: ");
        String z = sc.next(); //Heute ist ein schoener Tag

        System.out.println(z);

        sc.useDelimiter("-"); //Neues Trennzeichen festlegen
        System.out.print("Eingabe: ");
        z = sc.next(); //Das In-den-Tag-Hineinleben
        System.out.println(z);
    }
}
```

```
Eingabe: Heute ist ein schöner Tag
Heute
Eingabe: Das In-den-Tag-Hineinleben
ist ein schöner Tag
Das In
```

- Ein **Scanner**-Objekt (hier: **sc**) liest eine Zeichenkette stets nur bis zum ersten **Trennzeichen** (engl. *delimiter*) ein, der Rest wird **gepuffert**.
Idee: nächsten Befehl bzw. nächsten Parameterwert entgegennehmen
- Das Leerzeichen ist als Trennzeichen voreingestellt.
- Die Methode **UseDelimiter** erlaubt es, das Trennzeichen selbst festzulegen

Zeichenketten von Tastatur eingeben per Console

```
import java.io.*;

public class Eingabe {
    public static void main(String[] args) {

        System.out.print("Eingabe: ");
        Console c = System.console();
        String z = c.readLine();

        System.out.println("-----");
        System.out.println(z);
    }
}
```

- In der Java-Bibliothek **io** (*in-out*) Klasse **Console** verfügbar
- Konsolen-Objekte, die mit **System.console()** angelegt werden, nutzen Tastatur als Datenquelle
- Methode **readLine()** liest einen (beliebig langen) *Zeichenstrom* ein, bis Endezeichen (`'\n'` voreingestellt) erreicht ist und weist ihn direkt in voller Länge einer Zeichenkette zu.

Zahlenwerte in Zeichenketten konvertieren

```
public class Zeichenketten5 {  
    public static void main(String[] args) {  
  
        double guthaben = 123.45;  
  
        String z = String.valueOf(guthaben);  
  
        System.out.println(z);  
    }  
}
```

- Die Klasse **String** kann auch im Sinne einer *Methodenbibliothek* (wie z.B. **Math**) verwendet werden.
- Die Methode **valueOf** wandelt Variablen- oder Konstantenwerte beliebiger elementarer Datentypen (also **boolean**, **byte**, **char**, **short**, **int**, **long**, **float** und **double**) in entsprechende Zeichenketten um.

Zeichenketten aneinanderhängen

```
public class Konkatenation {  
    public static void main(String[] args) {  
  
        String s1 = "Hallo";  
        String s2 = "liebe";  
        String s3 = "Freunde";  
  
        String s4 = s1 + " " + s2 + " " + s3;  
  
        System.out.println(s4);  
    }  
}
```

Hallo liebe Freunde

- Der Operator `+` ist auf `String`-Objekten definiert und bewirkt das Aneinanderhängen („*Konkatenieren*“) der Zeichenketten von links nach rechts.
- Seien `s1` und `s2` Zeichenketten. Statt `s1 = s1 + s2`; kann man alternativ auch schreiben `s1 = s1.concat(s2)`;

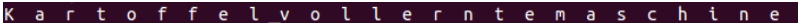
Länge einer Zeichenkette bestimmen

```
public class Zeichenketten6 {  
    public static void main(String[] args) {  
  
        String z = "Kartoffelvollerntemaschine";  
  
        int anz = z.length();  
  
        System.out.println("Anzahl Zeichen: " + anz); // 26  
    }  
}
```

- **length()** ermittelt die *Anzahl Zeichen* in einer Zeichenkette (Stringlänge)
- Die Länge einer leeren Zeichenkette (" ") ist **0**
- Beachte: Im Gegensatz zu Feldern ist **length()** auf Zeichenketten eine Methode und kein Attribut. Die runden Klammern **()** müssen daher geschrieben werden.

Zeichen an der i -ten Position einer Zeichenkette

```
public class Zeichenketten7 {  
    public static void main(String[] args) {  
  
        String z = "Kartoffelvollerntemaschine";  
        char c;  
  
        for (int i = 0; i < z.length(); i++)  
        {  
            c = z.charAt(i);  
            System.out.print(c + " ");  
        }  
    }  
}
```



- Das Zeichen an der i -ten Position einer Zeichenkette liefert die Methode **charAt**
- Die Positionen sind mit **0** beginnend bis **length() - 1** nummeriert
- Positionsangaben außerhalb dieses zulässigen Bereichs führen zu einem Laufzeitfehler

Zeichenketten auf Gleichheit vergleichen

```
public class Zeichenketten8 {  
    public static void main(String[] args) {  
        String z1 = "Ich habe liebe Genossen."  
        String z2 = "Ich habe Liebe genossen."  
  
        if (z1.equals(z2))  
        {  
            System.out.println("Beide Strings identisch.");  
        }  
        if (z1.equalsIgnoreCase(z2))  
        {  
            System.out.println("Identisch ohne Beachtung von Gross-/Kleinschreibung.");  
        }  
        z1 = z1.toLowerCase();  
        z2 = z2.toLowerCase();  
        if (z1.equals(z2))  
        {  
            System.out.println("In Kleinschreibung identisch.");  
        }  
    }  
}
```

Identisch ohne Beachtung von Gross-/Kleinschreibung.
In Kleinschreibung identisch.

- **equals** vergleicht zwei Zeichenketten auf Gleichheit und gibt **true** bzw. **false** zurück
- **equalsIgnoreCase** vernachlässigt beim Vergleich die Groß- und Kleinschreibung
- **toLowerCase** wandelt alle Großbuchstaben in Kleinbuchstaben um
- **toUpperCase** wandelt alle Kleinbuchstaben in Großbuchstaben um

Zeichenketten alphabetisch vergleichen

```
public class Zeichenketten9 {  
    public static void main(String[] args) {  
        String r = "rot";  
        String b = "blau";  
        String t = "rot";  
  
        System.out.printf("Vergleich: blau < rot: %d\n", b.compareTo(r));  
        System.out.printf("Vergleich: rot > blau: %d\n", r.compareTo(b));  
        System.out.printf("Vergleich: rot == rot: %d\n", r.compareTo(t));  
    }  
}
```

```
Vergleich: blau < rot: -16  
Vergleich: rot > blau: 16  
Vergleich: rot == rot: 0
```

- **compareTo** *vergleicht* zwei Zeichenketten bzgl. ihrer lexikographischen Anordnung
- **x.compareTo(y)**; liefert **0**, wenn **x == y** (beide Strings gleich)
- **x.compareTo(y)**;
liefert einen Wert **< 0**, wenn **x < y** (x alphabetisch vor y)
- **x.compareTo(y)**;
liefert einen Wert **> 0**, wenn **x > y** (x alphabetisch hinter y)
- Gelieferter Wert entspricht der Unicode-Differenz des ersten nicht übereinstimmenden Zeichens
(hier: **16** bzw. **-16**, da **r** im Unicode **16** Zeichen hinter **b**)

Alphabet durch Unicode-Werte gegeben

32	0	48	@	64	P	80	`	96	p	112	
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
\$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[91	k	107	{	123
,	44	<	60	L	76	\	92	l	108		124
-	45	=	61	M	77]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

Unicode ordnet jedem Zeichen eineindeutig eine Zahl zu

Deutsche Landeshauptstädte alphabetisch sortieren

Landeshauptstädte deutscher Bundesländer



stepmap.de 

www.stepmap.de

Selectionsort auf Zeichenketten (SelectionsortStrings.java)

```
public static void main(String[] args)
{
    String[] datenfeld = {"Kiel", "Schwerin", "Hamburg", "Bremen",
                          "Berlin", "Potsdam", "Hannover", "Magdeburg",
                          "Duesseldorf", "Dresden", "Erfurt", "Wiesbaden",
                          "Mainz", "Saarbruecken", "Stuttgart", "Muenchen"};
    //von Nord nach Sued

    int i;

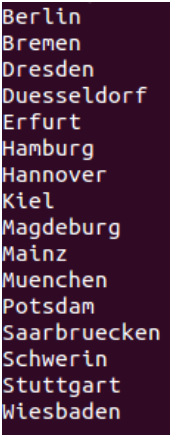
    selectionsort(datenfeld); //in-place sortieren
    for(i = 0; i < datenfeld.length; i++)
    {
        System.out.println(datenfeld[i]); //Feldelemente ausgeben
    }
}
```

Anfangsanordnung der Städte von Nord nach Süd

Selectionsort auf Zeichenketten (SelectionsortStrings.java)

```
public static void selectionsort(String[] a) //aufsteigend sortieren
{
    int i, k, min;
    String t;

    for(i = 0; i < a.length-1; i++)
    {
        min = i;
        for(k = i+1; k < a.length; k++) //Kleinstes Element im noch
            //zu sortierenden Teilfeld finden
            if(a[k].compareTo(a[min]) < 0)
            {
                min = k;
            }
        if (i != min)
        {
            t = a[min];           //Vertausche kleinstes Element mit
            a[min] = a[i];       //Anfangselement im noch zu
            a[i] = t;            //sortierenden Teilfeld
        }
    }
}
```



Berlin
Bremen
Dresden
Duesseldorf
Erfurt
Hamburg
Hannover
Kiel
Magdeburg
Mainz
Muenchen
Potsdam
Saarbruecken
Schwerin
Stuttgart
Wiesbaden

Ziel: Laufzeitfehler abfangen und behandeln

In Java-Programmen können *Fehler* enthalten sein, die der *Compiler nicht erkennt* oder *nicht erkennen kann* und die während der *Laufzeit* zu einem *Programmabbruch* führen.

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 26
    at java.lang.String.charAt(String.java:658)
    at Test.main(Test.java:9)
```



**Laufzeitfehler
im Programm
aufgefangen und behandelt**

```
Index ausserhalb des zulaessigen Bereichs  
... und das Programm laeuft weiter.
```

Fehlerbedingte Programmabbrüche sind unschön und sollten vermieden werden.

Laufzeitfehler führt zum Programmabbruch

```
public class Test {  
    public static void main(String[] args) {  
  
        String z = "Kartoffelvollerntemaschine";  
        char c;  
  
        for (int i = 0; i < z.length() + 5; i++)  
        {  
            c = z.charAt(i);  
            System.out.println(c + " ");  
        }  
    }  
}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 26  
    at java.lang.String.charAt(String.java:658)  
    at Test.main(Test.java:9)
```

Während der Programmlaufzeit wird in der Zeichenkette **z** auf Zeichenpositionen zugegriffen, die nicht existieren. Der erste dieser Zugriffe verursacht einen Laufzeitfehler, wodurch das Programm beendet wird.

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

ArithmeticException Division durch 0

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden
- IOException** Eingabe-Ausgabe-Fehler

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden
- IOException** Eingabe-Ausgabe-Fehler
- NullPointerException** Objekt nicht vorhanden

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden
- IOException** Eingabe-Ausgabe-Fehler
- NullPointerException** Objekt nicht vorhanden
- NumberFormatException** unpassendes Zahlenformat

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden
- IOException** Eingabe-Ausgabe-Fehler
- NullPointerException** Objekt nicht vorhanden
- NumberFormatException** unpassendes Zahlenformat
- OutOfMemoryException** nicht genug Speicherplatz

Es gibt viele verschiedene Arten von Laufzeitfehlern

Tritt während der Programmausführung ein Fehler auf, so wird eine *Ausnahme* (engl. *exception*) ausgelöst, die unbehandelt einen Programmabbruch nach sich zieht.

Mögliche Ausnahmen (Auswahl)

- ArithmeticException** Division durch 0
- FileNotFoundException** Datei nicht vorhanden
- IndexOutOfBoundsException** Feldelement nicht vorhanden
- IOException** Eingabe-Ausgabe-Fehler
- NullPointerException** Objekt nicht vorhanden
- NumberFormatException** unpassendes Zahlenformat
- OutOfMemoryException** nicht genug Speicherplatz

Laufzeitfehler werden jeweils durch diejenige Klasse ausgelöst, in denen sie auftreten.

Abfangen und Behandeln von Laufzeitfehlern

```
try
{
    // Quelltext, der eine Ausnahme auslösen kann
}
catch ( ... )
{
    // Quelltext zum Behandeln der Ausnahme
}

... // nachfolgender Quelltext

// Quelltextabarbeitung geht ganz normal weiter,
// denn die Ausnahme wurde behandelt
```

Idee: Den Java-Code, der eine Ausnahme auslösen kann, „*versuchsweise*“ ausführen (in einen **try**-Block einschließen) und dort möglicherweise auftretende Ausnahmen in nachfolgenden **catch**-Blöcken „*abfangen*“, also identifizieren und behandeln. Danach läuft das Programm einfach weiter.

Abfangen und Behandeln von Laufzeitfehlern

```
public class Test {  
    public static void main(String[] args) {  
  
        String z = "Kartoffelvollerntemaschine";  
        char c;  
  
        try  
        {  
            for (int i = 0; i < z.length() + 5; i++)  
            {  
                c = z.charAt(i);  
                System.out.println(c + " ");  
            }  
        }  
        catch (IndexOutOfBoundsException e)  
        {  
            System.err.println("Index ausserhalb des zulaessigen Bereichs");  
        }  
        System.out.println("... und das Programm laeuft weiter.");  
    }  
}
```

```
Index ausserhalb des zulaessigen Bereichs  
... und das Programm laeuft weiter.
```

Merke

- Tritt in einem **try**-Block eine Ausnahme auf, so *springt* die Programmabarbeitung in den dahinterstehenden **catch**-Block, und es wird geprüft, ob dort eine Behandlung der Ausnahme hinterlegt ist

Merke

- Tritt in einem **try**-Block eine Ausnahme auf, so *springt* die Programmabarbeitung in den dahinterstehenden **catch**-Block, und es wird geprüft, ob dort eine Behandlung der Ausnahme hinterlegt ist
- Der zur *Behandlung dieser Ausnahme* vorgegebene Quelltext im **catch**-Block wird abgearbeitet und anschließend die *normale Ausführung* des Programms unmittelbar hinter dem **catch**-Block *fortgesetzt*

Merke

- Tritt in einem **try**-Block eine Ausnahme auf, so *springt* die Programmabarbeitung in den dahinterstehenden **catch**-Block, und es wird geprüft, ob dort eine Behandlung der Ausnahme hinterlegt ist
- Der zur *Behandlung dieser Ausnahme* vorgegebene Quelltext im **catch**-Block wird abgearbeitet und anschließend die *normale Ausführung* des Programms unmittelbar hinter dem **catch**-Block *fortgesetzt*
- Tritt eine Ausnahme auf, die nicht abgefangen wird, so bricht das Programm mit einem Laufzeitfehler ab

Merke

- Tritt in einem **try**-Block eine Ausnahme auf, so *springt* die Programmabarbeitung in den dahinterstehenden **catch**-Block, und es wird geprüft, ob dort eine Behandlung der Ausnahme hinterlegt ist
- Der zur *Behandlung dieser Ausnahme* vorgegebene Quelltext im **catch**-Block wird abgearbeitet und anschließend die *normale Ausführung* des Programms unmittelbar hinter dem **catch**-Block *fortgesetzt*
- Tritt eine Ausnahme auf, die nicht abgefangen wird, so bricht das Programm mit einem Laufzeitfehler ab
- Auf einen **try**-Block dürfen *mehrere* **catch**-Blöcke für jeweils unterschiedliche Ausnahmen unmittelbar hintereinander folgen

Merke

- Tritt in einem **try**-Block eine Ausnahme auf, so *springt* die Programmabarbeitung in den dahinterstehenden **catch**-Block, und es wird geprüft, ob dort eine Behandlung der Ausnahme hinterlegt ist
- Der zur *Behandlung dieser Ausnahme* vorgegebene Quelltext im **catch**-Block wird abgearbeitet und anschließend die *normale Ausführung* des Programms unmittelbar hinter dem **catch**-Block *fortgesetzt*
- Tritt eine Ausnahme auf, die nicht abgefangen wird, so bricht das Programm mit einem Laufzeitfehler ab
- Auf einen **try**-Block dürfen *mehrere* **catch**-Blöcke für jeweils unterschiedliche Ausnahmen unmittelbar hintereinander folgen
- Hinter den **catch**-Blöcken darf auch noch ein **finally**-Block folgen, der sowohl ausgeführt wird, wenn es im **try**-Block zu einer Ausnahme kam als auch dann, wenn keine Ausnahme auftrat (Nützlich bei Ressourcenfreigaben, z.B. Datei schließen)

Eigene Ausnahmen auslösen

```
// ...  
  
int age = Integer.parseInt(sc.next()); // Alter von Tastatur einlesen  
  
if (age < 0)  
{  
    throw new IllegalArgumentException("Kein Alter < 0 erlaubt");  
}  
  
// ...
```

Mit **throw** in Java-Quelltexten gezielt selber erzeugte Ausnahmen auslösen

- Das Schlüsselwort **throw** („*wirf*“) erzeugt eine Ausnahme, indem ein neues Objekt einer Exception-Klasse angelegt wird.
- Der Ausnahme kann eine Zeichenkette zur Fehlerbeschreibung mitgegeben werden
- Auslösen der Ausnahme beendet Programmabarbeitung an der Stelle
- Danach sucht die *Java Virtual Machine* (Laufzeitumgebung) einen passenden **catch**-Block
- Existiert ein passender **catch**-Block, so erfolgt dort die Fehlerbehandlung, und die Programmabarbeitung wird dahinter fortgesetzt, anderenfalls mit einem Laufzeitfehler abgebrochen.

Behandlung von Ausnahmen weiterdelegieren

```
// ...  
  
public static void meineMethode(int a) throws ArithmeticException  
{  
    int b = a / 0; // Löst eine ArithmeticException aus,  
                  // deren Behandlung an die aufrufende  
                  // Methode weitergegeben wird  
}  
  
// ...
```

Mit **throws** Ausnahmebehandlung weiterdelegieren

- Hinter jedem Methodenkopf können nach dem Schlüsselwort **throws** – jeweils durch Komma getrennt – Ausnahmen angegeben werden.
- Wird während der Methodenabarbeitung eine dieser Ausnahmen ausgelöst, so schaut die *Java Virtual Machine* nach behandelndem **catch**-Block in übergeordneter (aufrufender) Methode.
- Im obigen Beispiel also in der Methode, die **meineMethode** aufruft.
- Diese darf ihrerseits die Ausnahmebehandlung weiterdelegieren.
- Findet sich innerhalb der Aufrufhierarchie ein passender **catch**-Block, so wird die Ausnahme behandelt, ansonsten bricht das Programm ab.

Arbeiten mit Textdateien in Java



Textdateien anlegen, schreiben und lesen mittels
Java-Programm

Eine Textdatei anlegen und beschreiben

```
import java.io.*;

public class TextdateiSchreiben
{
    public static void main(String[] args) throws IOException
    {
        String dateiname = "dateiname.txt";

        String dateiinhalt = "Dummytext ohne tiefe Bedeutung.";

        //Dateiobjekt anlegen
        File f = new File(dateiname);

        //Objekt zum gepufferten Schreiben anlegen und initialisieren
        BufferedWriter bw = new BufferedWriter(new FileWriter(f));

        //Textdatei beschreiben
        bw.write(dateiinhalt);

        //Textdatei schliessen
        bw.close();
    }
}
```

Eine Textdatei öffnen und lesen

```
import java.io.*;

public class TextdateiLesen {
    public static void main(String[] args) throws IOException {
        String dateiname = "dateiname.txt";
        String dateiinhalte = "";

        //Dateiobjekt anlegen
        File f = new File(dateiname);

        //Objekt zum gepufferten Lesen anlegen und initialisieren
        BufferedReader br = new BufferedReader(new FileReader(f));

        //Inhalt der Textdatei zeilenweise einlesen
        while (true)
        {
            String zeile = br.readLine();
            if (zeile == null) {break;} // Dateiende erreicht
            dateiinhalte = dateiinhalte + zeile;
        }

        //Textdatei schliessen
        br.close();

        System.out.println(dateiinhalte);
    }
}
```

Länge einer Textdatei ermitteln

```
import java.io.*;

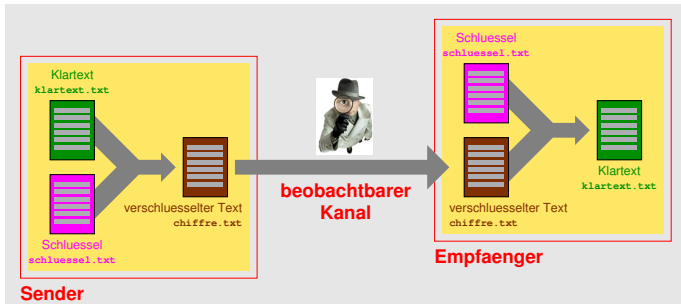
public class TextdateiLaenge {
    public static void main(String[] args) throws IOException {
        String dateiname = "dateiname.txt";
        long laenge; // in Byte

        //Dateiobjekt anlegen
        File f = new File(dateiname);

        laenge = f.length();

        System.out.printf("Laenge der Datei %s: %d Bytes\n", dateiname, laenge);
    }
}
```


Textverschlüsselung mit One-Time-Pad



- *einfaches* und bei richtiger Anwendung *beweisbar sicheres* Kryptoverfahren
- Schlüssel: zufällige Abfolge von Zeichen, genauso viele Zeichen wie Klartext. Gleicher Schlüssel für Sender und Empfänger
- Sender und Empfänger haben Schlüssel *auf Vorrat* ausgetauscht
- Schlüssel nur *einmal* („one time“) zum Ver- und Entschlüsseln verwendet

Klartext

Zu verschlüsselnder Text. Hier: 800 Zeichen in 10er-Blöcken

```

Lorem ipsu m dolor si t amet, co nsetetur s adipscing
elitr, sed diam nonum y eirmod t empor invi dunt ut la
bore et do lore magna aliquyam e rat, sed d iam volupt
ua. At ver o eos et a ccusam et justo duo dolores et
ea rebum. Stet clita kasd guber gren, no s ea takimat
a sanctus est Lorem ipsum dolo r sit amet . Lorem ip
sum dolor sit amet, consetetur sadipscing elitr, sed
diam nonu my eirmod tempor inv idunt ut l abore et d
olore magn a aliquya m erat, se d diam vol uptua. At
vero eos e t accusam et justo d uo dolores et ea rebu
m. Stet cl ita kasd g ubergren, no sea tak imata sanc
tus est Lo rem ipsum dolor sit amet. Lore m ipsum do
lor sit am et, conset etur sadip scing elit r, sed dia
m nonumy e irmod temp or invidun t ut labor e et dolor
e magna al iquyam era t, sed dia m voluptua . At vero
eos et acc usam et ju sto duo do lores et e a rebum.
```

Klartext als *ASCII-Textdatei* aus *druckbaren Zeichen* (ASCII-Werte 32 bis 127)

Schlüsseltext

Zeichenkette aus zufälligen Zeichen. Hier: 800 Zeichen in 10er-Blöcken

*U%{YLOWq"	□_ \$7>MyAx4	?H/C69/9nS	p, '=1Y~Hnw	.yv~59=H:
\fGtvObJ?*	@zrq ji@O	m~(Z)oprpf?	T0~cZz\$]u-	Of2oQ>:~T'
LOWi?G[%0	U/7ud\[K,U	QfS%P]' {nV	Au{E{QRp6.	KMyx"J ^ upl
EJl f[S(PY	>KsFEV6{H}	zjHX_8E%96	ul=b\V'O6:	^ 7H□Z' 7Lq.
^ '#J0Ah}_3	dmlw\$m?\$Gg	[f3Ls8\#yc6	sR"F7;3#3W	r{XYW># Sz
: ^ pQ-\$1s[1	&~}yfkP%xs	xEo5!ADNeu	A@bg>dL]3□	-Xw!P+S pZ
&{PjI5' uBq	3CUW33V' M[\-. C\.LVP	G:: jOm@0x1	Sj ^ f q?MmX
_ { .fKu _[I	aHAC!hY],'	JF31(p:g:0	D-q\p ^ F/;g	FWkL8.GE, Y
\KMy;SWMn})<' O}, aMK	cy\$VAK#:vT	e.Qj[m;=95	Ee0zTzaFOM
{'!'soxNf)	#B ^ h'uzWo+	jg@0l}goH}	7v_E\in□+L	gSBa*1fVMK
9&H!gix Xe	eASdlK?a, j	@.'Vz B:)#	%"C.z(;{w'	Bg nQ>'g8z
dz4v\$XFI@d	\{?S"ZWZfk	\$dS\ ^ 7W-{j	HAJm @dU(;	X![nF?A"xy
0%tm5 ~6jk	2t+QXyX{43	izs*"@G[4@	{2wf)ZH2H	sE oyTXtGW
%'m2s!snTj	Tq.: '%#pwI	>L\$qK- ^ ~ q	T[cEn□ r~%	n77JZnou9S
7=S(L'0mrB	ln4n1L{/M;	yu ^ ,S2U YX	PFJ39F!%5d	dn)g4}APDp
n@d9Rr9,nB	R+&@TF'Z\$u	_le} &aYxy	K2b= zr%&3	ylZG□bt{Yx

Zufallszeichen *gleichverteilt* und *unabhängig voneinander*

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Beispiel: $t[i] = 'I'$ (ASCII-Wert 76) und $s[i] = '*'$ (42)
 $c[i] = 'v'$ (86)

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Beispiel: $t[i] = 'l'$ (ASCII-Wert 76) und $s[i] = '*'$ (42)
 $c[i] = 'v'$ (86)

- $t[i]$ und $s[i]$ sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Beispiel: $t[i] = 'l'$ (ASCII-Wert 76) und $s[i] = '*'$ (42)
 $c[i] = 'v'$ (86)

- $t[i]$ und $s[i]$ sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Beispiel: $t[i] = 'I'$ (ASCII-Wert 76) und $s[i] = '*'$ (42)
 $c[i] = 'v'$ (86)

- $t[i]$ und $s[i]$ sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127
- Es gibt insgesamt 96 verschiedene druckbare Zeichen.

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei $t[i]$ das *Klartextzeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < t.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Chiffre-Zeichen* $c[i]$ wie folgt:

$$c[i] = ((t[i]-32) + (s[i]-32)) \% 96 + 32$$

Beispiel: $t[i] = 'l'$ (ASCII-Wert 76) und $s[i] = '*'$ (42)
 $c[i] = 'v'$ (86)

- $t[i]$ und $s[i]$ sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127
- Es gibt insgesamt 96 verschiedene druckbare Zeichen.
- Aus einem beliebigen Klartextzeichen kann je nach Schlüsselzeichen jedes Chiffre-Zeichen werden.

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei $c[i]$ das *Chiffre-Zeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < c.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Klartextzeichen* $t[i]$ wie folgt:

$$t[i] = \begin{cases} (c[i] - s[i]) \% 96 + 128 & \text{falls } s[i] > c[i] \\ c[i] - (s[i] - 32) & \text{sonst} \end{cases}$$

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei $c[i]$ das *Chiffre-Zeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < c.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Klartextzeichen* $t[i]$ wie folgt:

$$t[i] = \begin{cases} (c[i] - s[i]) \% 96 + 128 & \text{falls } s[i] > c[i] \\ c[i] - (s[i] - 32) & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei $c[i]$ das *Chiffre-Zeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < c.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Klartextzeichen* $t[i]$ wie folgt:

$$t[i] = \begin{cases} (c[i] - s[i]) \% 96 + 128 & \text{falls } s[i] > c[i] \\ c[i] - (s[i] - 32) & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.
- Es wird der gleiche Schlüssel sowohl zum Ver- als auch zum Entschlüsseln genutzt. Das heißt, die $s[i]$ stimmen jeweils überein.

Entschlüsseln

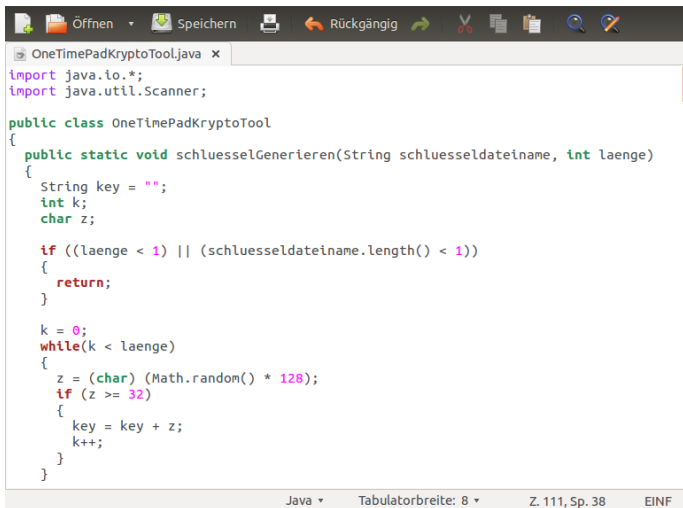
Es wird zeichenweise entschlüsselt. Sei $c[i]$ das *Chiffre-Zeichen* und $s[i]$ das *Schlüsselzeichen* an der i -ten Position der jeweiligen Zeichenkette ($0 \leq i < c.length()$ und $0 \leq i < s.length()$). Dann ergibt sich das *Klartextzeichen* $t[i]$ wie folgt:

$$t[i] = \begin{cases} (c[i] - s[i]) \% 96 + 128 & \text{falls } s[i] > c[i] \\ c[i] - (s[i] - 32) & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.
- Es wird der gleiche Schlüssel sowohl zum Ver- als auch zum Entschlüsseln genutzt. Das heißt, die $s[i]$ stimmen jeweils überein.
- Durch das Mapping in den ASCII-Bereich der druckbaren Zeichen zwischen 32 und 127 erscheinen die Berechnungsvorschriften zum Ver- und Entschlüsseln etwas aufgebläht.

Programmdemo One-Time-Pad-KryptoTool

OneTimePadKryptoTool.java Quelltext auf Veranstaltungsw Webseite verfügbar



```
import java.io.*;
import java.util.Scanner;

public class OneTimePadKryptoTool
{
    public static void schluesselGenerieren(String schluesseldateiname, int laenge)
    {
        String key = "";
        int k;
        char z;

        if ((laenge < 1) || (schluesseldateiname.length() < 1))
        {
            return;
        }

        k = 0;
        while(k < laenge)
        {
            z = (char) (Math.random() * 128);
            if (z >= 32)
            {
                key = key + z;
                k++;
            }
        }
    }
}
```

Java ▾ Tabulatorbreite: 8 ▾ Z. 111, Sp. 38 EINF

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf = 0 und Zahl = 1)

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf = 0 und Zahl = 1)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf = 0 und Zahl = 1)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.
- Ist der Wert zwischen 32 und 127, hat man den ASCII-Wert eines Zufallszeichens.
- Werte zwischen 0 und 31 werden ignoriert.

Wie lassen sich zufällige Zeichen erzeugen?

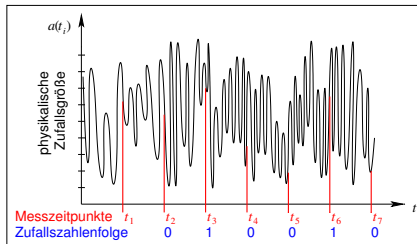
- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf = 0 und Zahl = 1)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.
- Ist der Wert zwischen 32 und 127, hat man den ASCII-Wert eines Zufallszeichens.
- Werte zwischen 0 und 31 werden ignoriert.

Problem: Diese Vorgehensweise ist sehr umständlich und sehr aufwendig.

Rauschgrößenmessung

z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:
 $a(t_j) \geq a(t_{j-1}) \rightarrow \text{Ausgabe } y(t_j) = 1$
 $a(t_j) < a(t_{j-1}) \rightarrow \text{Ausgabe } y(t_j) = 0$



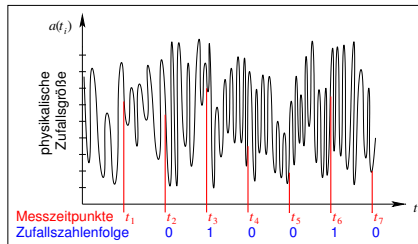
Rauschgrößenmessung

z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:
 $a(t_j) \geq a(t_{j-1}) \rightarrow \text{Ausgabe } y(t_j) = 1$
 $a(t_j) < a(t_{j-1}) \rightarrow \text{Ausgabe } y(t_j) = 0$

Vorteile

- gute statistische Eigenschaften
- keine inhärente Reproduzierbarkeit



Rauschgrößenmessung

z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

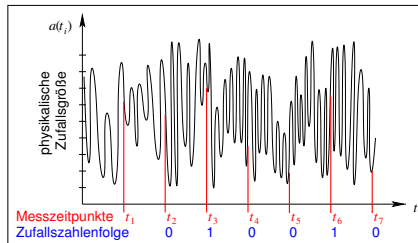
- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:
 $a(t_i) \geq a(t_{i-1}) \rightarrow \text{Ausgabe } y(t_i) = 1$
 $a(t_i) < a(t_{i-1}) \rightarrow \text{Ausgabe } y(t_i) = 0$

Vorteile

- gute statistische Eigenschaften
- keine inhärente Reproduzierbarkeit

Nachteile

- Auswirkungen von Messfehlern
- Verfügbarkeit und maximale Abtastrate abhängig von physikalischer Zufallsgröße



$x^2 \bmod n$ Generator nach Blum/Shub

- **Prinzip**

Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \pmod{4}$,
 $0 < s < pq$, $\text{ggT}(s, pq) = 1$

Startwert: $z_0 = s^2 \bmod (pq)$

Rekursion: $z_i = z_{i-1}^2 \bmod (pq)$

PZZ-Folge: $r_i = z_i \bmod 2$ Es gilt: $r_i \in \{0, 1\}$

$x^2 \bmod n$ Generator nach Blum/Shub

- **Prinzip**

Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \pmod{4}$,
 $0 < s < pq$, $\text{ggT}(s, pq) = 1$

Startwert: $z_0 = s^2 \bmod (pq)$

Rekursion: $z_i = z_{i-1}^2 \bmod (pq)$

PZZ-Folge: $r_i = z_i \bmod 2$ Es gilt: $r_i \in \{0, 1\}$

- **maximale Periodenlänge**

pq

$x^2 \bmod n$ Generator nach Blum/Shub

- **Prinzip**

Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \pmod{4}$,
 $0 < s < pq$, $\text{ggT}(s, pq) = 1$

Startwert: $z_0 = s^2 \bmod (pq)$

Rekursion: $z_i = z_{i-1}^2 \bmod (pq)$

PZZ-Folge: $r_i = z_i \bmod 2$ Es gilt: $r_i \in \{0, 1\}$

- **maximale Periodenlänge**

pq

Vorteile

- perfekter Pseudozufallszahlengenerator
- leichte Implementierbarkeit
- leichte Wahl geeigneter Parameterbelegungen

$x^2 \bmod n$ Generator nach Blum/Shub

- **Prinzip**

Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \pmod{4}$,
 $0 < s < pq$, $\text{ggT}(s, pq) = 1$

Startwert: $z_0 = s^2 \bmod (pq)$

Rekursion: $z_i = z_{i-1}^2 \bmod (pq)$

PZZ-Folge: $r_i = z_i \bmod 2$ Es gilt: $r_i \in \{0, 1\}$

- **maximale Periodenlänge**

pq

Vorteile

- perfekter Pseudozufallszahlengenerator
- leichte Implementierbarkeit
- leichte Wahl geeigneter Parameterbelegungen

Nachteile

- nur ein Bit pro Rekursionsschritt \rightarrow langsam
- kleine Periodenlänge