

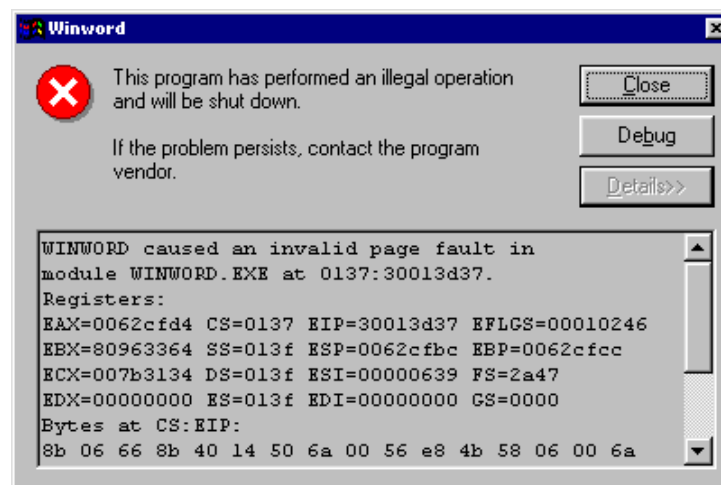
# Ausnahmen



Eine Ausnahme (Exception) ist ein Fehler oder ein nicht geplantes Ereignis, das während der Ausführung eines Programms vorkommt und dessen normalen Ablauf stört.

# Ausnahmen

Wenn Programme nicht auf Ausnahmesituationen reagieren können, führt das zu den von den Anwendern gefürchteten Abstürzen.



In Java wurde von Anfang an die Behandlung von Ausnahmen in die Sprache integriert.

Java ermöglicht eine "**weiche**" Landung nach Fehlern.

# Ausnahmen und Fehler

Ausnahmen  
und  
Fehler

**Fehler (Error)** in Java ist ein nicht reparierbarer Laufzeitfehler oder ein Hardware-Problem, das zum Absturz des Programms führt.

**Ausnahmen (Exceptions)** sind meistens keine eigentlichen Fehler, sondern unerwartete Fälle, auf die das Programm reagieren muss.

# Warum Ausnahmebehandlung?

Nehmen wir an, wir bieten eine Klasse *Menge* mit entsprechenden Einfüge-, Lösch- und Such-Operationen.

Was passiert, wenn bei einer Suchoperation ein Element nicht gefunden wird?

## 1. Lösung:

Wir können unsere Suchoperation so definieren, dass die **null** Konstante zurückgegeben wird, wenn ein Element nicht gefunden wird.

### Probleme:

Der Benutzer muss dieses Verhalten genau kennen, und wenn er das vergisst, führt das oft zum unbeliebten **NullPointerException**-Fehler mit entsprechendem Programmabsturz.


## 2. Bessere Lösung:

Wir zwingen den Benutzer, den Fall zu behandeln, sonst wird sein Programm nicht übersetzt.

# Ausnahmebehandlung in Java

Die Ausnahme wird Bestandteil der Methoden-Signatur

Beispiel:



```
public Object suche ( Object x ) throws NotFoundException {  
    . . .  
}
```

Das hier bedeutet, dass innerhalb des Methodenrumpfs eine Ausnahme mit diesem Namen stattfinden kann.

Ein Ausnahme-Objekt der Klasse **NotFoundException** wird innerhalb der Methode erzeugt, falls das gesuchte Objekt nicht gefunden wird.

# Ausnahmebehandlung in Java

**Ausnahmen** ( ***Exceptions*** ) sind unerwartet auftretende **Laufzeitfehler**.

Eine **Ausnahme** in Java ist ein **Objekt**, das eine Instanz der Klasse ***Throwable*** (oder einer ihrer Unterklassen) ist.

- Beispiele:
- Division durch **0**  
( ***ArithmeticException*** )
  - Lesen über Arraygrenzen hinweg  
( ***IndexOutOfBoundsException*** )
  - Lesen über das Ende einer Datei hinaus  
( ***EOFException*** )

## **throw**-Anweisung

**throw new** Exception ( "Schlechte Nachrichten" );

```
public class Time{
    private int seconds, minutes, hours;
    ...
    public void setSeconds( int secs) throws Exception
    {
        if( secs<0 || secs>59 )
            throw new Exception( "Falscher ..." );
        else
            this.seconds = secs;
    }
    ...
}
```

# Ausnahmebehandlungsschema

Ausnahmen müssen grundsätzlich abgefangen oder weitergereicht werden.

Ein Ausnahme-Objekt  
wird innerhalb einer  
Methode erzeugt.

**auslösen**

**throw** -Anweisung

**abfangen**

**try-catch** -Anweisung

In der aufrufenden  
Methode muss die  
Ausnahme behandelt oder  
weitergeleitet werden.

**weiterreichen**  
**throws**-Klausel



# Programmieren von Ausnahme-Klassen

```
public class NegativeUeberweisungException extends Exception {  
  
    double betrag;  
    String dieb;  
    String opfer;  
    // Konstruktor  
    public NegativeUeberweisungException(String opfer,String dieb,double betrag)  
    {  
        super ( "Negative Überweisungen sind nicht erlaubt!" );  
        this.betrag = betrag;  
        this.opfer = opfer;  
        this.dieb = dieb;  
    }  
} // end of class NegativeUeberweisungException
```

# Ausnahme-Erzeugung

Ein Ausnahme-Objekt wird mit der **new**-Anweisung erzeugt und mit der **throw**-Anweisung geworfen.

```
...  
public void ueberweisung ( Bankkonto empfaenger, double betrag )  
    throws NegativeUeberweisungException  
{  
    if ( betrag > 0 ) {  
        abheben( betrag );  
        empfaenger.zahlen( betrag );  
    } else {  
        throw new NegativeUeberweisungException  
            ( empfaenger.name, this.name, betrag );  
    }  
}  
...
```

# Das Grundkonzept

Das Grundkonzept der Behandlung von Ausnahmen in Java folgt dem Schema:

<b>throw</b>	Auslösen
<b>try</b>	Ausprobieren
<b>catch</b>	Auffangen

Beim Auftreten einer Ausnahme wird die Ausführung des Programms unterbrochen und ein **Ausnahmeobjekt** "geworfen"

( mit **throw** ).

Es wird nach einer passenden Ausnahmebehandlung gesucht, die das Ausnahmeobjekt "auffängt" ( **catch** ).

Wird keine Ausnahmebehandlung gefunden, wird das Programm abgebrochen.

# Behandlung von Ausnahmen

```
...
```

```
try
```

```
{
```

```
    riskyBusiness ( );
```

```
}
```

```
catch ( SomeExceptionType e )
```

```
{
```

```
    // Handle bad stuff
```

```
}
```

```
...
```

Methode, die  
Ausnahmefehler  
auslösen kann

Referenz des  
Ausnahmefehlers, der  
"gefangen" werden muss.

Reaktion auf den  
Ausnahmefehler

## Ausnahme-Behandlung

Wenn eine Methode, die Ausnahmen erzeugen kann, benutzt wird, muss diese in einer **try-catch**-Anweisung umschlossen werden, um mögliche Ausnahmefehler zu behandeln, anderenfalls muss die Behandlung delegiert werden.

## Ausnahme-Behandlung

```
...  
Bankkonto bk1 = new Bankkonto( "Benjamin", 5000 );  
Bankkonto bk2 = new Bankkonto( "Tobias", 2000 );  
  
...  
try {  
    bk1.ueberweisung (bk2,-1000);  
} catch ( NegativeUeberweisungException nue ) {  
    System.out.println( nue.getMessage() );  
    System.out.println( nue.betrag );  
    System.out.println( "Opfer = " + nue.opfer );  
    System.out.println( "Dieb = " + nue.dieb );  
}  
  
...
```

Negative Überw ... !  
-1000.0  
Opfer = Tobias  
Dieb = Benjamin

# Exceptions

Der aufrufende "unsichere" Code wird mit einem **try**- Block umschlossen.

Die darin auftretenden **Exceptions** können mit 1 oder mehr **catch** (**Exception e**) - Blöcken abgefangen werden.

Jeder **Exception-Typ e** wird spezifisch behandelt.

In dem Ausnahme-Objekt kann **zusätzliche Information** zwecks **Konfliktbereinigung** übergeben werden.

Um den genauen Ort des Fehlers festzustellen, kann man **e.printStackTrace()** aufrufen.

# Funktionsprinzip

Mit **try** wird eine Momentaufnahme des Zustands der Umgebung des Programms gemacht.

## Stack + Register

Dann wird der nach **try** in {...} angegebene Block betreten.

Tritt eine Programmausnahme auf, so wird der Stack nach **catch**-Routinen durchsucht.

**try/catch** kann nicht mehr als den Stack zurückrollen und Inhalte von Registern wiederherstellen.



## try-catch-Anweisung

```
riskyMethod() {  
    wirft einen  
    Ausnahmefehler  
}
```

```
try  
{  
    riskyMethod ( );  
}  
catch ( NullPointerException ne ) {  
    // Anweisungen für NullPointerException..  
}  
catch ( ArithmeticException ae ) {  
    // Anweisungen für ArithmeticEx..  
}  
catch ( IndexOutOfBoundsException ie ) {  
    // Anweisungen für IndexOutOfBou..  
}  
catch ( Exception e ) {  
    // behandelt alle anderen Fehler  
}
```

Exception

## Ausnahmebehandlung delegieren

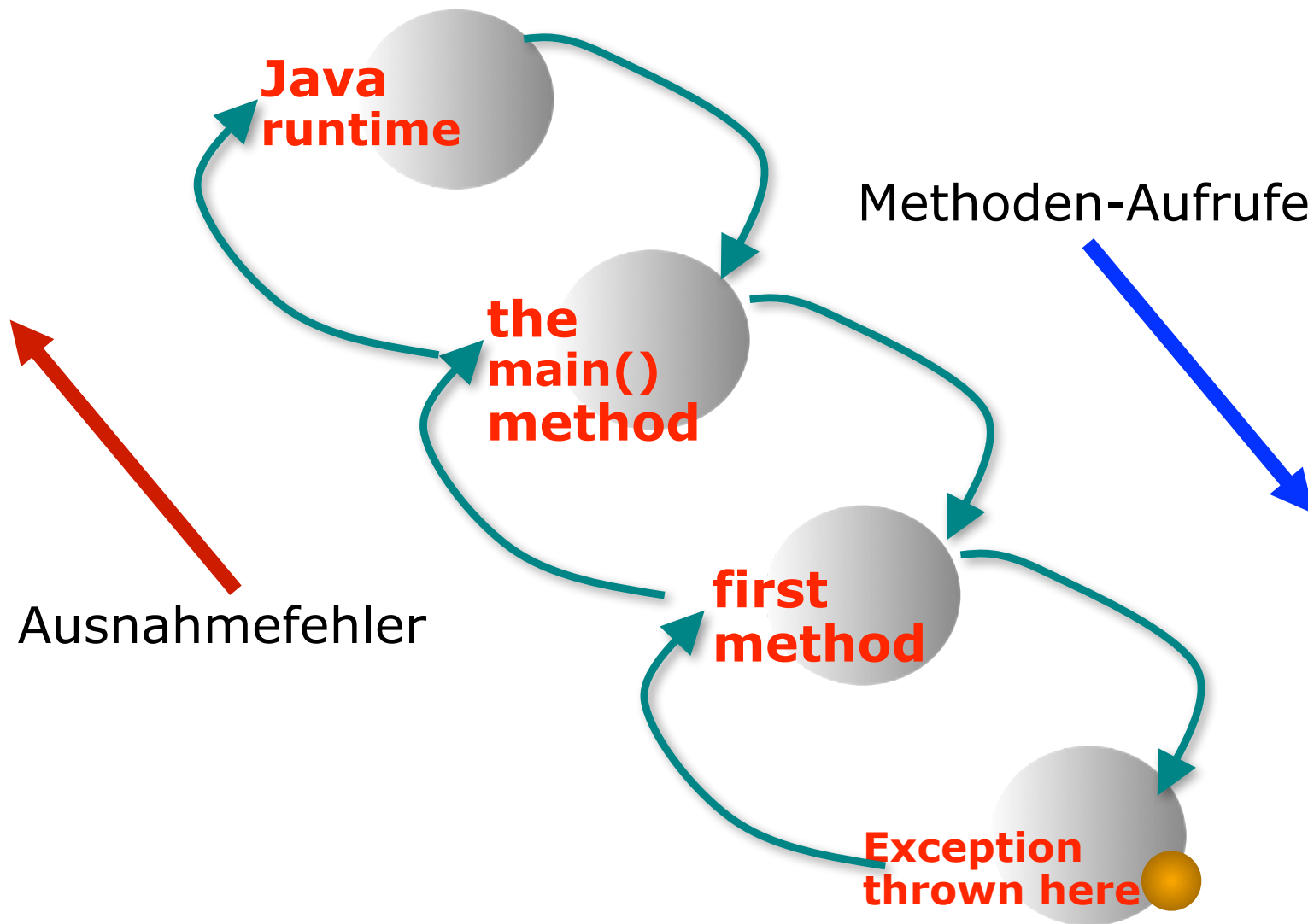
Wenn innerhalb einer Methode die Ausnahmebehandlung nicht gemacht wird, muss wenigstens die Behandlung an die aufrufenden Methoden, mit Hilfe der **throws**-Anweisung, delegiert werden.

Beispiel:

```
...  
void zahlt (Bankkonto zahler, Bankkonto empfaenger, double geld)  
    throws NegativeUeberweisungException  
{  
    zahler.ueberweisung ( empfaenger, geld );  
}  
...
```

Wenn alle Methoden die Ausnahmebehandlung delegieren, kann ein Laufzeitfehler verursacht werden.

# Laufzeitfehler



```
main {  
  method_1( );  
}
```

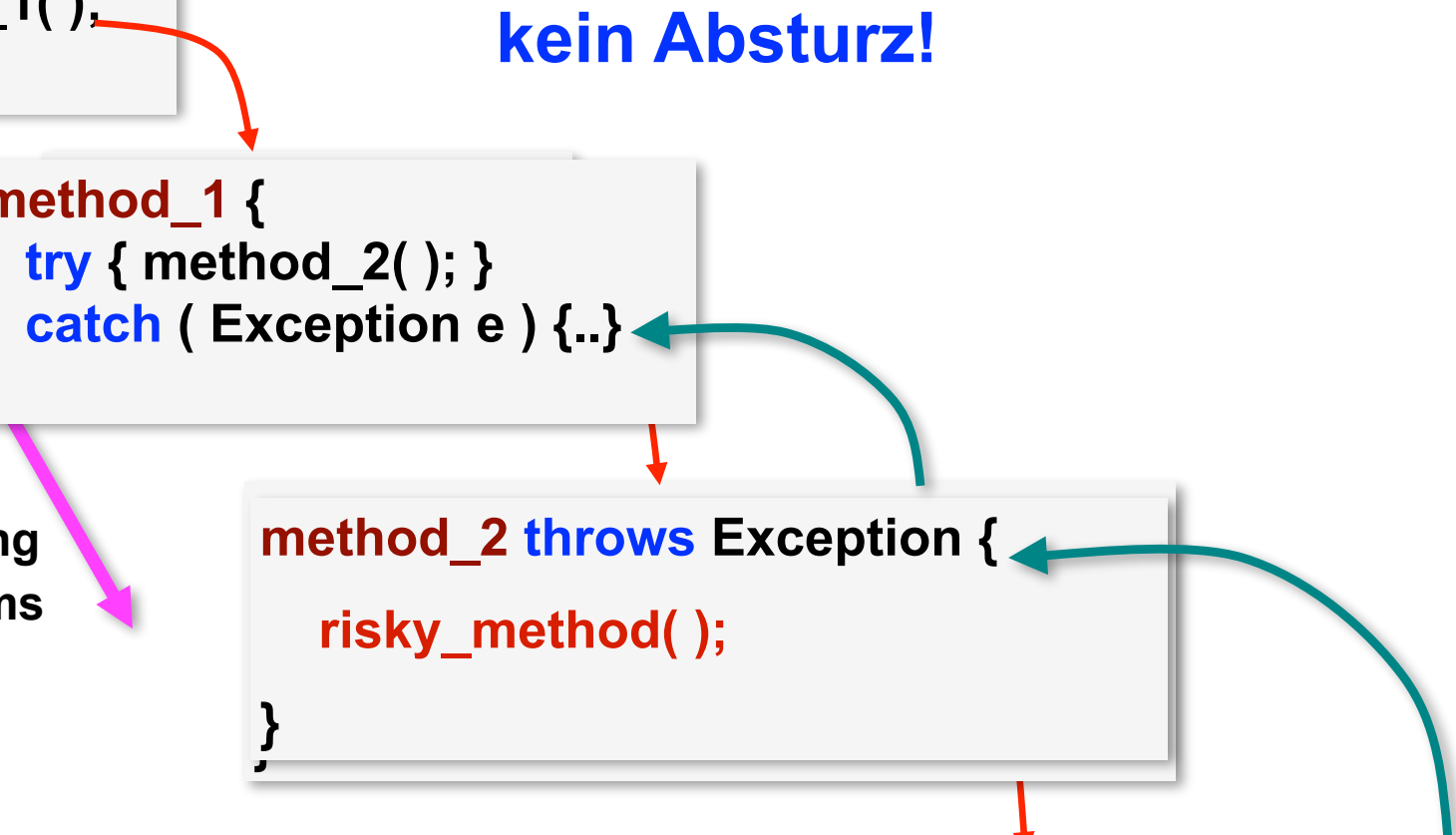
kein Absturz!

```
method_1 {  
  try { method_2( ); }  
  catch ( Exception e ) { .. }  
}
```

Die Ausführung  
des Programms  
geht weiter.

```
method_2 throws Exception {  
  risky_method( );  
}
```

```
risky_method( ) throws Exception {  
  if ( somethingWrong )  
    throw new Exception( "Feh..." )  
}
```



Absturz nur aus dem Erdgeschoss!

**main** throws Exception

```
{  
  method_1( );  
}
```

**method\_1** throws Exception

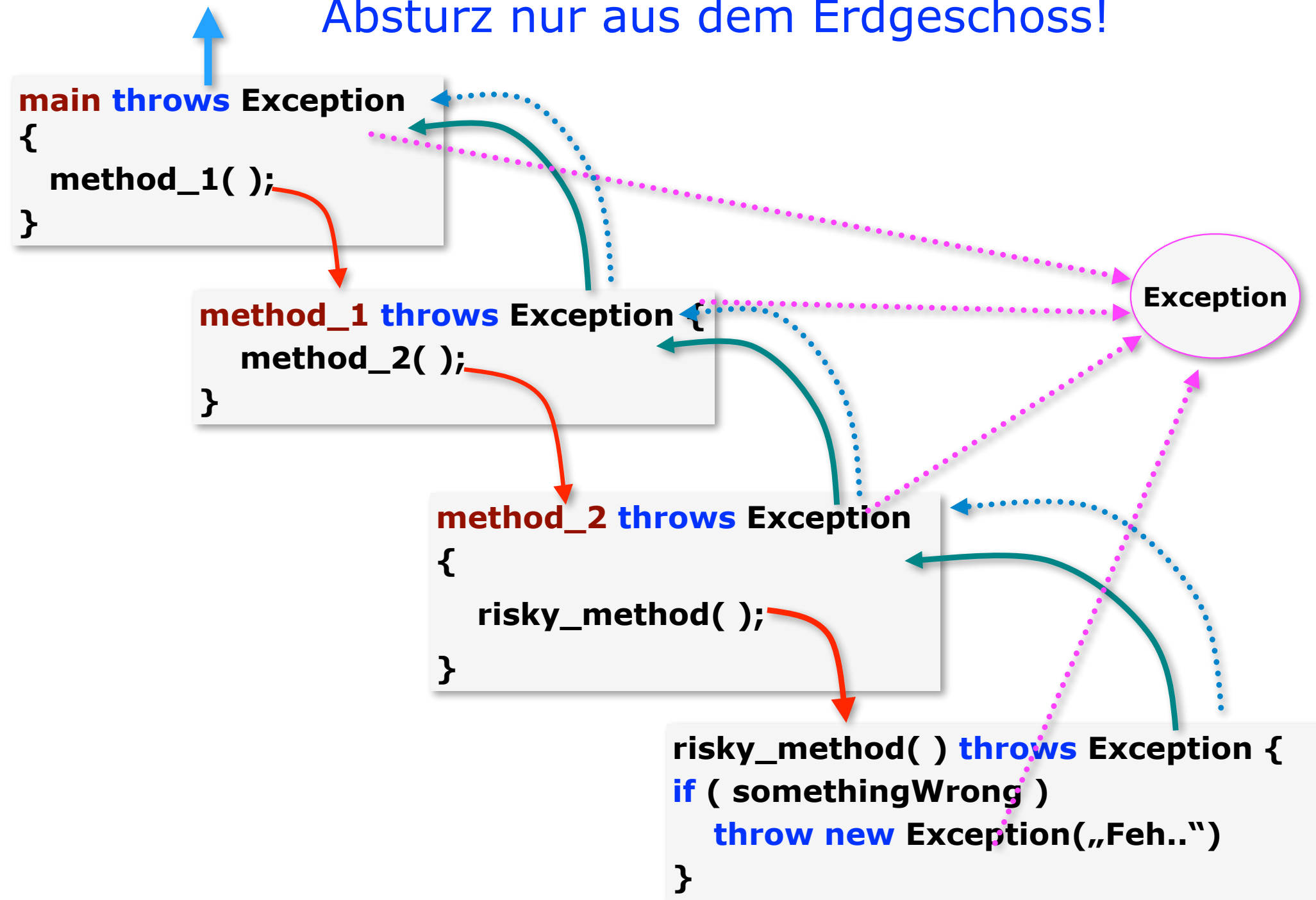
```
{  
  method_2( );  
}
```

**method\_2** throws Exception

```
{  
  risky_method( );  
}
```

```
risky_method( ) throws Exception {  
  if ( somethingWrong )  
    throw new Exception(„Feh..“)  
}
```

Exception



## throws-Klausel

```
public int readInt()  
    throws IOException, NumericFormatException  
{  
    int ch;  
    int integer = 0;  
    String s = "";  
    while ( ( ch = System.in.read() ) != '\n' )  
        { s = s + (char) ch; }  
    integer = Integer.parseInt(s);  
    return integer;  
}
```

kann eine  
NumericFormatException !!  
auslösen

kann eine  
IOException !!  
auslösen

## try-catch-finally-Anweisung

Es besteht zusätzlich die Möglichkeit, zu jedem **try**-Block Anweisungen zu definieren, die auch im Ausnahmefall sicher ausgeführt werden.

- Beispiele:
- Objekte in einen sicheren Zustand bringen
  - Dateien schließen

Dazu wird ein **finally**-Block vereinbart, der diese Anweisungen enthält:

```
try { ... }  
catch (  $E_1$   $e_1$  ) { ... }  
:  
catch (  $E_n$   $e_n$  ) { ... }  
finally { ... }
```

**wird immer ausgeführt!**

## **try-catch-finally**-Anweisung

Es gab **keine Ausnahme!**

Es gab **eine Ausnahme** und diese wurde **in** einer **catch-Anweisung** behandelt !

Es gab **eine Ausnahme**, aber **keine catch-Anweisung** war für diesen Fehler zuständig !

**In allen 3 Fällen wird die Finally-Anweisung ausgeführt!**



# Exception-Klasse

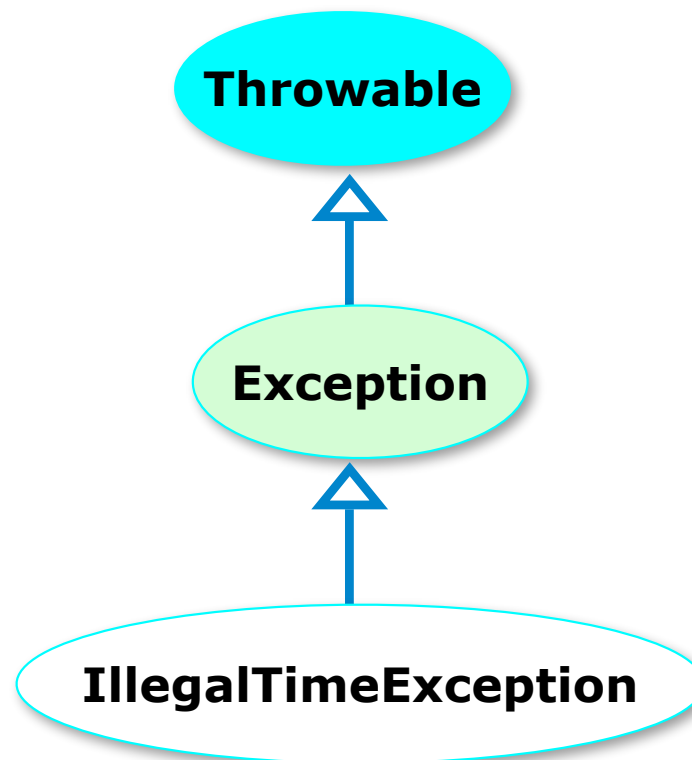
## Konstruktoren:

```
Exception( )  
Exception( String s )
```

Einige Methoden, die von der Klasse `java.lang.Throwable` vererbt werden, sind:

```
getMessage()  
printStackTrace()  
toString()  
fillInStackTrace()  
getLocalizedMessage()
```

## Definition neuer Ausnahmen



Neue Ausnahmeklassen werden als Unterklasse der Klasse **Exception** definiert.

# Definition neuer Ausnahme-Klassen

```
public class IllegalTimeException extends Exception {  
    Time wrongTime;  
    public IllegalTimeException ( String reason ) {  
        super ( reason );  
    }  
    public IllegalTimeException (String reason, Time wrongTime) {  
        super ( reason );  
        this.wrongTime = wrongTime;  
    }  
    ...  
}
```

## RuntimeException

Exceptions, die Unterklassen von **RuntimeException** sind, sind sogenannte "**unchecked exceptions**" – sie müssen nicht in einer **throws**-Klausel deklariert werden, und der Übersetzer verlangt keine expliziten Ausnahmebehandlungen.

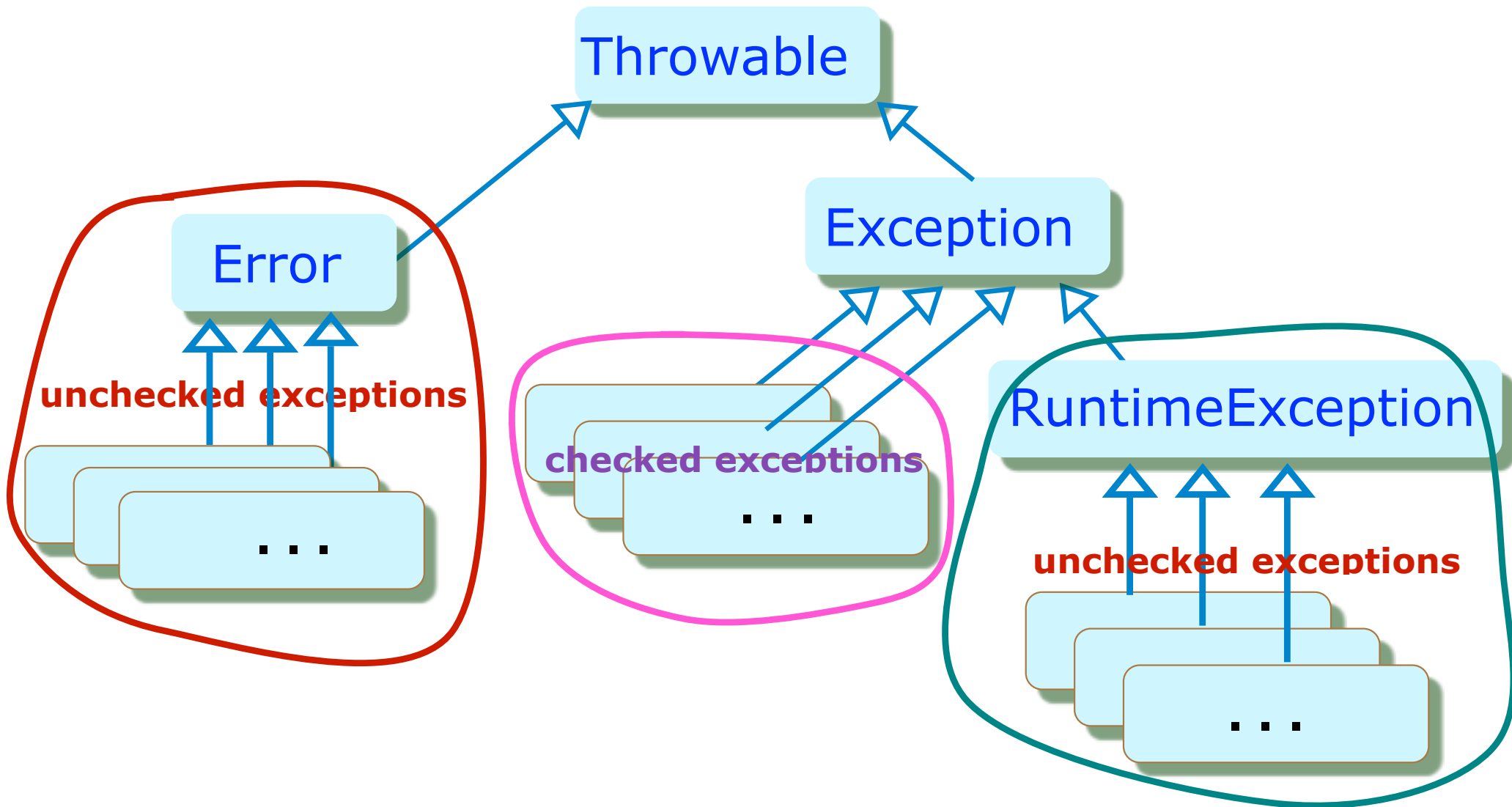
**RuntimeExceptions sind Ausnahmen, die durch gutes Programmieren vermeidbar sind!!**

## "checked exceptions"

Ausnahmen, die im Allgemeinen nicht vermeidbar sind, aber vom Programmierer behandelt werden können bzw. sollten.

Sie müssen in einer **throws**-Klausel deklariert werden.

## Verschiedene Arten von Ausnahmen



## Gemeinsame Oberklassen

Ist die Behandlung aller möglichen Ausnahmen jeweils gleich, kann man sie ggf. auch zusammenfassen. Z.B. haben Ausnahmeobjekte bei der Ein-/Ausgabe eine gemeinsame Oberklasse **IOException**.

```
try {  
    ...  
} catch ( IOException e ) { ... }
```

# Regeln zum Umgang mit Ausnahmen

- Ausnahmebehandlung nicht zur Behandlung normaler Programmsituationen einsetzen !
- Ausnahmebehandlung nicht in zu kleinen Einheiten durchführen !
- Auf keinen Fall Ausnahmen "abwürgen" oder "ignorieren" z.B. durch triviale Fehlermeldungen !
- Ausnahmen zu propagieren ist keine Schande!

# Schlüsselwörter in Java

abstract ✓	default ✓	if ✓	private ✓	throw ✓
boolean ✓	do ✓	implements ✓	protected ✓	throws ✓
break ✓	double ✓	import ✓	public ✓	<b>transient</b>
byte ✓	else ✓	instanceof ✓	return ✓	try ✓
case ✓	extends ✓	int ✓	short ✓	void ✓
catch ✓	final ✓	interface ✓	static ✓	<b>volatile</b>
char ✓	finally ✓	long ✓	super ✓	while ✓
class ✓	float ✓	<b>native</b>	switch ✓	true ✓
const ✓	for ✓	new ✓	<b>synchronized</b>	false ✓
continue ✓	goto ✓	package ✓	this ✓	null ✓

nur reserviert

Literalkonstanten