Es gibt die zu prüfenden (checked) und die nicht (zwangläufig) zu prüfenden (unchecked) Ausnahmen (Exceptions). Dies sind Fehler, die den Programmablauf stören. Sie sind logischer oder funktionaler Natur. Obgleich alle Exceptions während der Laufzeit eintreten, gibt es die Klasse Exception (checked) und die davon erbende Klasse RunTimeException (unchecked). Wir sind hier also schon nicht mehr auf Compilerebene.

**Exception-Handling**

Realisiert wird das durch einen try-catch(-finally)-Block. Dabei gibt es einen try-Block, ein oder mehrere Catch-Blöcke und optional einen finally-Block; letzterer ist dabei optional und wird immer ausgeführt, egal ob ein Fehler eintrat oder nicht. Bei mehreren catch-Blöcken muss darauf geachtet werden, dass die konkreteste Exception ganz oben steht und jede abstraktere weiter unten. So ist sichergestellt, dass Code nicht unerreichbar wird, weil ein anderer catch-Block im fälschlicherweise zuvorkommt. (Compiler-Error)

Es ist theoretisch (syntaktisch) auch möglich einen try Block nur mit einem finally-Block oder einer Ressourceninitialisierung um Kopf des try-Blocks (z.B. try(new Scanner(){}) anbei fehlerfrei zu kompilieren, aber es ist in etwa so sinnvoll, wie ein Auto ohne Bremsen.

Alles, was einen Fehler auslösen könnte, kommt in den try-Block. Dies kann entweder der Aufruf von Funktionen sein, die explizit Exceptions werfen oder das Ausführen einer Operation, bei der implizit das Auftreten eines Fehlers zu erwarten ist. Tritt ein Fehler auf, wird die Ausführung des try-Blocks unterbrochen und nächstpassende catch-Block in der Abfolge ausgelöst. Ist man sich unsicher, welcher Fehler eintreten wird, kann man die oberste Klasse Exception einzig oder ganz unten angeben, um alle abzufangen. Wenn möglich sollten auftretende Fehler aber bestmöglich spezifiziert werden, um Fehler besser zu identifizieren, zu lokalisieren und somit das Debugging ungemein unterstützen.

Ist der erreichte catch-Block nicht der Ort oder Abschnitt, an welchem der Fehler behandelt werden soll, kann dieser auch weitergereicht werden, sofern es eine darüberliegende Methode gibt, welche selbst Exceptions abfängt und behandeln kann. Dies lässt sich vergleichen mit einer Murmelbahn, heiße Kartoffel oder Baseballspielen.  
Das Weiterreichen passiert zum Beispiel über throw new *Exception*(„“) (oder eine andere).

**Beispiel eines try-catch-finally-Blocks**

try(optionale Ressourceninitialisierung)

{ ein oder mehrere Statments und/oder Methodenaufrufe, die Exceptions auslösen }

catch (SpecificException | OtherException exceptionVar) //Man kann auch mehrere in eine Stufe packen

{ System.err.println(exceptionVar.getMessage); } //dabei gibt es aber nur eine Exceptionvariable

catch (ArithmeticException ae)

{ throw new ArithmeticException(„Nulldivision!“) }

catch (RuntimeException re) //Erbt von Ecxeption

{ System.err.println(re.getMessage); } //Textausgabe in rot bei err. statt out.

catch (Exception e) //letzter Ebene, falls alle anderen nicht gegriffen haben. (e als Bezeichner ist geläufig)

{ System.err.println(e.getMessage); } //Textausgabe in rot bei err. statt out.

finally { System.out.println(„Ich werde immer ausgeführt, egal was vorher passiert ist.“); }

**checked Exception**

Ein Fehler, der so gravierend ist, dass der Entwickler gezwungen ist, sich damit auseinanderzusetzen, wie er gehandhabt wird, wenn er eintritt. Sie sind erwartbar, wie das Fehlen einer benötigten Datei oder die Nichterreichbarkeit einer Datenbank. Dort greift dann das oben erläuterte Exception-Handling. Im Gegensatz zur RunTimeException müssen die (allgemeineren) Exceptions explizit und deklarativ gehandhabt werden.

Allgemeine Methodendeklaration (siehe sonst nochmal Tamagotchi-Projekt)

modifier returnType methodName(Paramtype param) throws CheckedDataException {

do some risky stuff to get the wanted value;

if (*couldNotGetDataFromFile || couldNotConnectToDatabase*)

{ throw new CheckedDataException(“Shit just got real!”); } //Konstr. setzt Message

//hier wird dann nicht mehr ausgeführt, wenn Exception geworfen

return wantedValue; //das hier also auch nicht }

Dann das Handling beim Aufruf:

try { Object obj = methodName(param); }

catch (CheckedDataException cde) {

System.err.println(cde.getMessage)

//oder auch wenn innerhalb einer weiteren Methode mit throws in der Signatur

thrown new ExceptionToCaller(„Hey Boss, something went wrong here!”); }

**unchecked Exception**

Dieser Fehler ist weniger gravierend und stört den Programmablauf zwar auch aber nicht so kritisch und gravierend, dass es nicht weitergehen kann, wenn es auftritt.

Wird bei einem Taschenrechner aber die Division durch 0 nicht als Arithmetic Exception gehandlet, so wird das Programm trotzdem abstürzen, wenn sie unbehandelt eintritt, aber damit zum Beispiel die Divisionsmethode eine werfen kann, muss es nicht in der Signatur stehen. Es ist also optional und man könnte auch hoffen, dass das nicht passiert.

Es kann hier schon reichen die fehlerhafte Operation zurückzumelden und den Nutzer zu einer anderen Division aufzufordern. Das Fehlen einer Datei hingegen verhindert den restlichen Ablauf des Programms, sofern dieser vom Inhalt fehlender Daten abhängig ist.

Hier Methodendeklaration ohne throws

public double division(double divident, double divisor){ //ohne throws ArithmeticExcetion

if (divisor == 0){

thrown new ArithmeticException(„Division durch 0!”) }

return dividend/divisor; }

Hier der Aufruf:

try {

double quotient = division(divident,divisor); }

catch (ArithmeticException ae)

{ System.err.println(ae.getMessage); }

Wenn ich mich recht erinnere ging das aber noch ein Ticken einfacher. Irgendwas war da mit Direktausgabe der Exception oder so.

Passt das so?

Das Ganze darf aber auch im „scope of occurance“ verbleiben:

public void rechnen(){ //ohne throws ArithmeticExcetion

double dividend = 5;

double divisor = 0;

double quotient = 0;

try { double quotient = division(divident,divisor); }

catch (ArithmeticException ae)

{ System.err.println(ae.getMessage); }

}

**Ergänzung am 17.01.2025:**

Nun konnte ich mir dank Dmytros Hilfe selber beweisen, wie die unchecked RuntimeException im Gegensatz zur checked Exception funktioniert.

Sie muss weder in der Signatur benannt werden mit throws

noch muss sie geworfen werden (throw) (nur nötig, wenn eigene Message gewollt)

Wird unbehandelt trotzdem zum Abbruch führen

Das Handling darf aber blind erfolgen, die Exception tritt quasi implizit auf.

Vergleich: Checked ist so brisant, dem Fänger muss vorher gesagt werden „Er Alter, da könnte ein Ball geflogen kommen. Halt Ausschau!“

Unchecked eher so: „Bleib mal hier stehen, falls ein Ball kommt.“

In beiden Fällen wird natürlich gefangen, aber da man nicht explizit bei unchecked werfen muss, ist man auch nicht gezwungen zu überlegen (checken) wann und/oder warum.

public class TesteRunTimeOhneThrow {  
 public static void main(String[] args) {  
 double quotient;  
  
 //unbehandelte (unchecked) Exception wird auftreten  
 //System.out.println(0/0);  
 //quotient = division(10, 0);  
  
 try {  
 //Exception wird auftreten, aber behandelt  
 quotient = *division*(10, 0);  
 System.*out*.println(quotient);  
  
 } catch (ArithmeticException ae) {  
 //"/ by zero", wenn ohne eigenes throw new ArithmeticException("")  
 System.*err*.println(ae.getMessage());  
 }  
 }  
  
  
 //warum int, wieso nicht double. Nun ja, double macht daraus "Infinity"  
 //da ich aber beweisend blind eine Exception auslösen wollte,  
 //war das nicht hilfreich.  
 public static int division(int dividend, int divisor) {  
 //throws ArithmeticException nicht nötig  
/\*  
 //auch optional weil unchecked, aber spezifischer  
 if (divisor == 0) {  
 throw new ArithmeticException("Division durch 0!");  
 }  
\*/  
 //gegenprobe checked, muss in signatur  
 //throw new Exception("Blabla");   
  
 return dividend / divisor;  
 }  
}