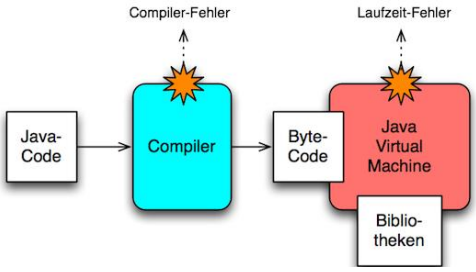
Compiler-Fehler

Java-Code wird durch den Compiler in ein "maschinenfreundliches" Format übersetzt, den Bytecode. Der Compiler kann beim Übersetzen bereits Fehler finden, die **Compiler-Fehler**. Solange diese Fehler nicht beseitigt sind, kann ein Programm nicht starten.

Beispiele für Fehler, die der Compiler abfängt, sind Syntaxfehler, aufgrund falsch geschriebener Befehle (wile statt while) oder vergessener Semikolons oder Klammern.

Es geht aber hier um die **Laufzeit-Fehler**, die dann auftreten, **wenn ein Programm läuft**.

Der Compiler kümmert sich auch um semantische Fehler, d.h. Code, der zwar syntaktisch stimmt, aber ein inhaltliches Problem hat.



Quelle: https://michaelkipp.de/java/

Der Unterschied zwischen Exception und Errors

Exceptions und Fehler sind in drei Kategorien gegliedert: geprüfte Exceptions, ungeprüfte Exceptions und Fehler.

Errors sind **schwere Ausnahmefehler**, die zur Laufzeit des Programms nicht behandelt werden können. Meist führen sie zum abrupten Programmende (Absturz).

Exceptions sind **Ausnahmefehler, die** während der Laufzeit eines Programms **"repariert" werden** können. Sie werden im weiteren Programmverlauf behandelt und vermeiden so z.B. einen Absturz des Programms. Das Programm bleibt weiter ausführbar.

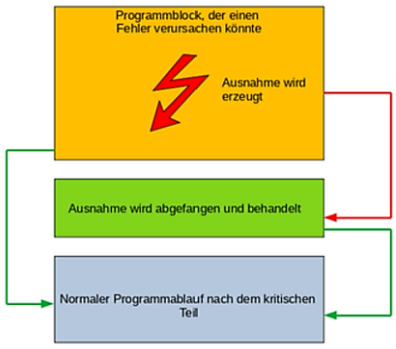


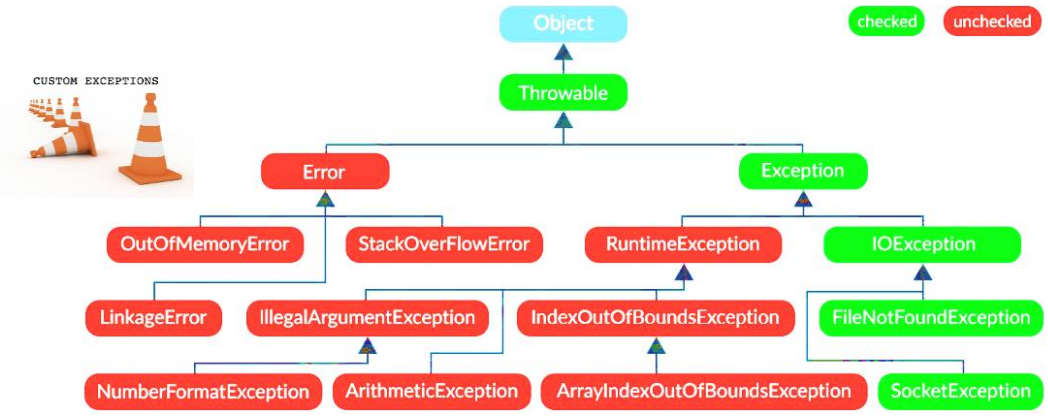
Abb. 1: Ablauf eines Java-Programms bei der Verwendung von Exceptions. © Christopher Olbertz

Geprüfte, benutzerdefinierte Exceptions müssen abgefangen (`throws` - Klausel) oder an den Aufrufer weitergeleitet werden. Bei geprüften Ausnahmen zwingt uns der Compiler dazu.

Bei **ungeprüften Exceptions** der Klassen `Error` und `RunTimeException` gibt es diese Pflicht nicht — falls wir jedoch eine `RunTimeException` nicht behandeln, führt dies zum Abbruch des ausführenden Programms.

Exception Hierarchie

Alle Exception - und Error-Typen sind Unterklassen der Klasse **Throwable.**



Oft tauchen Probleme auf, weil man nicht weiß, was eine Compiler Fehlermeldung bedeutet.

Die häufigsten Errors

Compiler-Errors:

- `cannot read <filename>`

Die Datei konnte nicht gelesen werden. Ist sie im CLASSPATH? Wurde der Dateiname richtig geschrieben?

- `cannot resolve symbol <symbol>`

Das Symbol (Methode, Klasse, Feld, Variable, Konstruktor) konnte nicht gefunden werden, d.h. es gibt dieses "Symbol" nicht. Ist es richtig geschrieben? ist evtl. die Parameterübergabe falsch gemacht?

- `<something> expected`

Es wird irgendein syntaktisches Symbol erwartet. Meist ist das Semikolon oder eine Klammer.

- `<variable> is already definded in ...`

Das bedeutet, dass man eine Variable zweimal in derselben Methode/Klasse deklariert hat; das darf man natürlich nicht, weil sich die Variablen sonst in die Quere kommen.

- `<class> is not abstract and does not override abstract method <methode> in <interface>`

Es müssen alle Methoden eines implementierten Interfaces überschrieben werden, wenn die Klasse nicht abstrakt ist. Beispiel:

- `<variable> might not have been intialized`

Das ist eine Warnung an den Programmierer, dass eine Variable nicht initialisiert worden ist. Der Fehler tritt nur bei lokalen Variablen auf. Eine Lösung ist z.B. der Variablen bei der Deklarierung den Wert null zuzuweisen.

- `unreported exception <exception>; must be caught or declared to be thrown`

In der angegebenen Zeile wird eine Methode aufgerufen, die eine Exception wirft. Diese Exception muss abgefangen werden oder mit "throws <exception>" in der Methodensignatur weitergeleitet werden.

- `missing methode body`

Ein typischer Anfängerfehler. Höchstwahrscheinlich wurde nach der Methodendeklaration ein Semikolon gesetzt.

- `possible loss of precision`

Zwei int Werte können bei der Division einen double-Wert ergeben, am besten auch als double deklarieren.

- `unclosed string literal`

bedeutet, dass ein String nicht geschlossen wurde (fehlende Anführungszeichen).

- `incompatible types`

Hier wird versucht einer Variablen den Wert einer anderen Variablen zuzuweisen, aber die beiden Typen sind nicht gleich Bsp.: String s = "Hello World"; int i = s; // falsch!!

- `<symbol> has private access in <class>`

Das angesprochene Symbol, also Variable oder Methode ist als privat deklariert, d.h. man kann von der aktuellen Klasse nicht darauf zugreifen.

Die RunTimeExceptions:

Runtime Exceptions:

- `NullPointerException`

Es wird auf ein nicht-initialisiertes oder ein leeres Objekt zugegriffen.

- `IllegalArgumentException`

Diese Exception wird geworfen, wenn ein übergebenes Argument an eine Methode oder Konstruktor nicht gültig ist. Folgendes ist ein Beispiel, welches eine Methode implementiert, die eine IllegalArgumentException werfen kann.

- `IndexOutOfBoundsException (bzw. ArrayIndexOutOfBoundsException)`

Dieser Fehler tritt häufig bei Arrays auf. Er zeigt an, dass man auf einen Index im Array gezeigt hat, der über den Grenzen des Arrays liegt.

- `StringIndexOutOfBounds`

Das ist der gleiche Fehler wie der obige, nur innerhalb eines Strings. Er tritt z.B. bei der Benutzung von substring() auf.

- `ArithmeticException`

wird geworfen, wenn ein Arithmetischer Ausnahmefall auftritt, z.B. bei der Division durch 0.

- `ClassCastException`

wird geworfen, wenn versucht wird ein Objekt in ein anderes zu casten, die beiden Typen aber nicht kompatibel sind.

Runtime Errors:

- `NoClassDefFoundError`

wird geworfen, wenn man versucht eine Klasse (per java.exe oder ClassLoader) zu laden, aber keine solche Klasse ist gefunden wurden. Ist die Klasse im CLASSPATH?
Dieser Fehler tritt bei Anfängern oftmals auf.

- `OutOfMemoryError`

wird geworfen, wenn die VirtualMachine versucht ein Objekt zu allokieren, aber kein Speicherplatz mehr der VM zur Verfügung steht.

- `StackOverflowError`

wird geworfen, wenn das Stack "überläuft". Das ist meistens der Fall, wenn eine Methode rekursiv aufgerufen wird, aber keine Endbedingung existiert.

- `UnsupportedClassVersionError`

wird geworfen, wenn versucht wird eine Klasse zu laden, das Format des Bytecodes aber nicht mit dem, der aktuellen Java Version übereinstimmt.

Wie fängt man Ausnahmen (exceptions) im Programm ab?

```
try {
            // Code der gesichert läuft
    }
catch (ExceptionKlassenName variablenname) {
        // Fehlerbehandlung
    }
      // optional
finally {
      // Programmcode der nach dem try-Block ausgeführt wird
}
```

Beispiel 1: eigene Exception

```
Bierbier = new Bier();
try    {
            // Fussballmannschaft betritt die Kneipe und trinkt bier
            bier.trinken(0.33);
            bier.trinken(0.5);
            bier.trinken(0.5);
            bier.trinken(0.5);
            bier.trinken(0.33);
            bier.trinken(0.2);
            bier.trinken(1);
            bier.trinken(0.5);
      }
         // Wenn das Bier alle ist - Der Fehler wird abgefangen
catch (BierProbleme e) {
            e.printStackTrace();
              // Es wird neues Bier nachgekauft! (10 Liter)
            bier.vorratInLiter = bier.vorratInLiter + 10;
            System.out.println();
            System.out.println("Es wurde Bier eingekauft!"+ bier.vorratInLiter + "l.");
       }
```

Beispiel 2: mathematische Exception

```
try     {
            result = 17/a; // Hier kann eine ArithmeticException ausgelöst werden
           System.out.println(result);
       }
catch (ArithmeticException e) { return 0;}
finally {                return 1;      }
```

Beispiel 3: Laufzeit- Exception

```
try    {
            for(int i=0;i<6;i++){robi.legeZiegel();schrittLinks();  };
            schrittLinks();
            for(int j=0;j<12;j++){robi.legeZiegel();schrittLinks(); };
       }
catch (RuntimeException e) {robi.gibMeldungAus("Kann Auftrag nicht ausführen");}
```