PAuFa

FEHLERBEHANDLUNG

Exception Handling

THEMENÜBERSICHT

- Assertions
- Exceptionhandling
- Try mit Ressourcen

ASSERTIONS

Annahmen

PAuFa

ASSERTIONS

- Mit Assertions besteht die Möglichkeit zur Laufzeit des Programmes bestimmte Eigenschaften zu überprüfen.
- Assertions werden für interne Prüfzwecke verwendet.

ASSERTIONS - SYNTAX

- Die Bedingung muss wie bei einer Verzweigung zu einem boolean ausgewertet werden.
- Klammern sind nicht notwendig, vereinfachen aber die Lesbarkeit.

```
int x = 10;
int y = 12;

assert (x>y);
```

```
int x = 10;
int y = 12;

assert x>y;
```

ASSERTIONS - SYNTAX

- Wird die Bedingung der Assert Anweisung zu false ausgewertet, wird ein AssertionError an die JVM weitergeleitet.

```
Exception in thread "main" java.lang.AssertionError
        at assertions.Demo01AssertionReallySimple.doStuff(Demo01AssertionReallySimple.java:34)
        at assertions.Demo01AssertionReallySimple.main(Demo01AssertionReallySimple.java:7)
```

ASSERTIONS - SYNTAX

- Damit Assertions von der JVM ausgewertet werden, müssen sie bei Programmstart durch ein Argument aktiviert werden.
- Standardmäßig sind Assertions deaktiviert.
- Argument für die Aktivierung: -ea oder –enableassertions
- Argument für die Deaktivierung: -da oder -disableassertions

ASSERTIONS - SYNTAX

- Die zweite Variante der Syntax für Assertions erlaubt es, einen String als Fehlermeldung hinzuzufügen.

```
int y = 10;
int x = 15;

assert (y>x): "y is not greater than x: x: " + x + " and y:" + y;
```

```
Exception in thread "main" java.lang.AssertionError: y is not greater than x: x: 15 and y:10
        at assertions.Demo02AssertionsSimple.doStuff01(Demo02AssertionsSimple.java:32)
        at assertions.Demo02AssertionsSimple.main(Demo02AssertionsSimple.java:22)
```

ANGEMESSENE VERWENDUNG VON ASSERTIONS

ASSERTIONS

- Assertions sollten verwendet werden:
  - In privaten Methoden, da wir den aufrufenden Code kontrollieren.
  - Im Switch Case(auch in öffentlichen Methoden), um Fälle zu prüfen, die garantiert nicht erreicht werden sollen.

ASSERTIONS

- Assertions sollten nicht verwendet werden:
  - um Argumente von öffentlichen Methoden zu prüfen, denn sie können von Code aufgerufen werden, den wir nicht kontrollieren.
  - um Kommandozeilenargumente zu validieren.
  - In beiden Fällen sind Exceptions vorzuziehen.
  - wenn Nebeneffekte auftreten, also die Logik des Programmes darf nicht davon abhängen, ob Assertions eingeschaltet sind oder nicht.

EXCEPTIONS

EXCEPTIONS

- Während der normalen Abarbeitung einer Methode kann zur Laufzeit ein abnormales Ereignis auftreten, das die normale Ausführung der Methode unterbricht. Ein solches abnormales Ereignis ist eine Exception (Ausnahme).

- Eine Exception kann beispielsweise ein arithmetischer Überlauf, ein Mangel an Speicherplatz, eine Verletzung der Array-Grenzen, etc. darstellen.

- Eine Exception stellt damit ein Laufzeitereignis dar, das zum Versagen einer Methode und damit zu einem Laufzeit-Fehler des Programms führen kann.

EXCEPTIONS

- Das Array hat eine Länge von drei, der letzte Index ist n-1.
- Beim Zugriff auf Index drei kommt es zu einem Laufzeitfehler, einer `ArrayIndexOutOfBoundsException`

```
int[] zahlen = new int[] { 1, 2, 3 };
int summe = 0;

for (int i = 0; i <= zahlen.length; i++) {
    summe += zahlen[i];
}

System.out.println("summe: " + summe);
```

EXCEPTIONS

- Die JVM beendet sich und es wird der Stacktrace in der Konsole ausgegeben.
- Der Stacktrace liefert Informationen darüber, welcher Fehler aufgetreten ist und den Weg, den die Exception durch die Methoden genommen hat.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at exceptions.pak01_throws01.FehlerArray.method02(FehlerArray.java:28)
        at exceptions.pak01_throws01.FehlerArray.method01(FehlerArray.java:22)
        at exceptions.pak01_throws01.FehlerArray.main(FehlerArray.java:19)
```

EXCEPTIONS ABFANGEN

Try – Catch

EXCEPTIONS ABFANGEN

- Um das Versagen der Methode und damit des Programmes im Fehlerfall zu verhindern, werden Try-Catch Blöcke verwendet, um eine Exception abzufangen.
- Zuerst muss der kritische Bereich im Code identifiziert werden.
- Im vorangegangenen Bsp. ist das die folgende Zeile:

```
int[] zahlen = new int[] { 1, 2, 3 };
int summe = 0;
for (int i = 0; i <= zahlen.length; i++) {
    summe += zahlen[i];
}
System.out.println("summe: " + summe);
```

Der kritische Bereich

EXCEPTIONS ABFANGEN

- Dieser kritische Bereich kann in einen Try-Catch Block eingepackt werden, damit sich das Programm von dem Fehler erholt und fortgeführt wird.
- Der kritische Bereich kommt in den Try-Block. Im Fehlerfall wird der Code im Catch-Block ausgeführt und das Programm wird durch den Fehler nicht beendet.

```
try
{
//kritischer Code, der einen Fehler verursachen kann
}
catch(Exception e) //Welcher Fehler soll abgefangen werden
{
//Code, der im Fehlerfall ausgeführt werden soll
}
```

EXCEPTIONS ABFANGEN

- Dieser kritische Bereich kann in einen Try-Catch Block eingepackt werden, damit sich das Programm von dem Fehler erholt und fortgeführt wird.
- Der kritische Bereich kommt in den Try-Block. Im Fehlerfall wird der Code im Catch-Block ausgeführt und das Programm wird durch den Fehler nicht beendet.

```
int[] zahlen = new int[] { 1, 2, 3 };
int summe = 0;
for (int i = 0; i <= zahlen.length; i++) {
    try {
        summe += zahlen[i];
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.printf("Es wurde versucht auf den Index %d zu greifen. %n", i);
    }
}
System.out.println("summe: " + summe);
```

EXCEPTIONS HIERARCHIE

EXCEPTIONS - HIERARCHIE

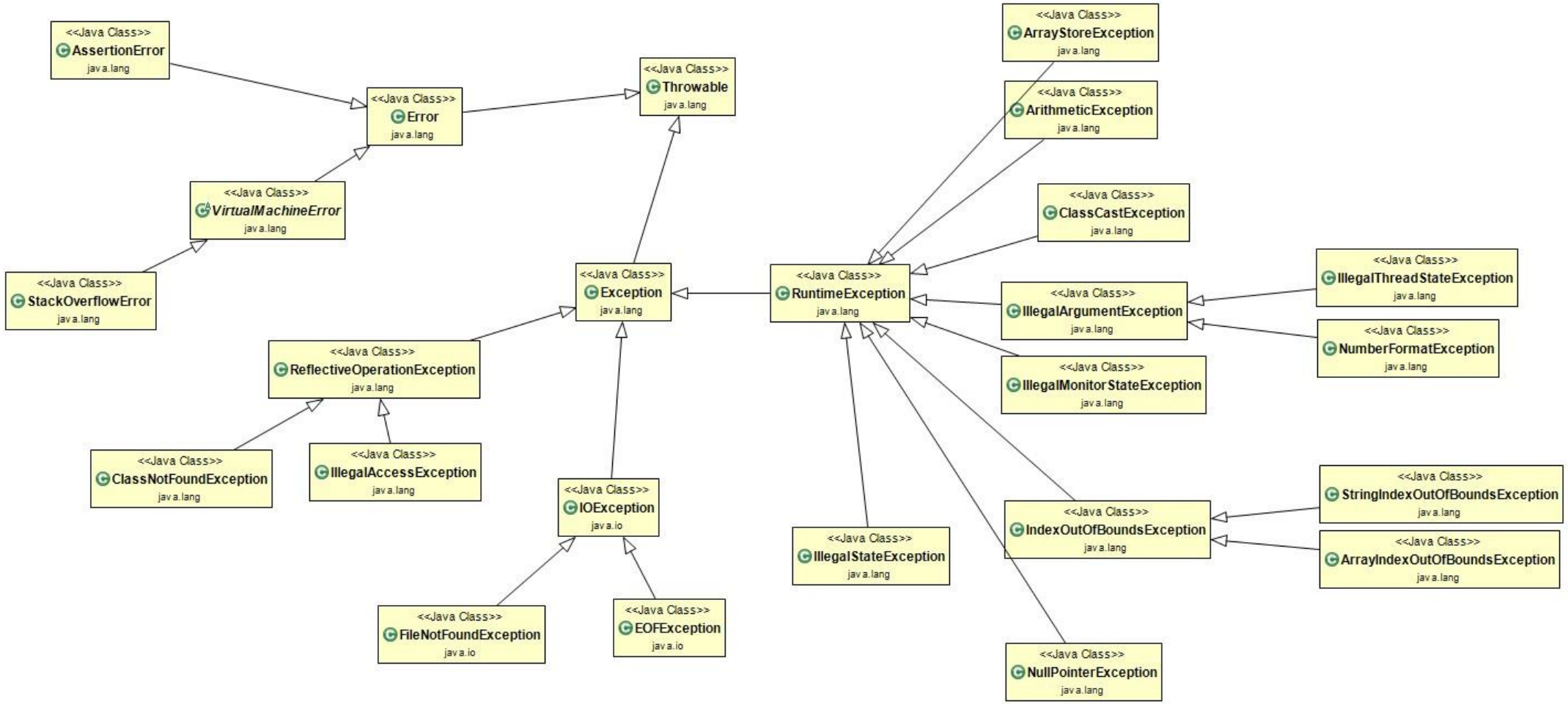
- Bei Exceptions handelt es sich um Objekte, die von der JVM im Fehlerfall erzeugt und geworfen werden. Sie kapseln die Informationen über den Fehler, z.B. über die Methode, in der der Fehler aufgetreten ist.

- Exceptions können auch vom Entwickler selbst erzeugt und geworfen werden, dazu später mehr.

- Die Oberklasse aller Exceptions ist Throwable. Throwable ist von Object abgeleitet.

- Es folgt eine Übersicht über die häufigsten Exceptions:

EXCEPTIONS - HIERARCHIE



FUNKTIONSWEISE

Try - Catch

PAuFa

FUNKTIONSWEISE TRY - CATCH

1. Das Exception Objekt wird von der JVM erzeugt und geworfen.
2. Das geworfene Exception Objekt wird vom Catch-Block gefangen. Es wird sofort in den Catch Block gesprungen. Code im Try Block nach der Exception wird nicht mehr ausgeführt. Innerhalb des Blockes kann mit dem Bezeichner auf die gefangene Exception zugegriffen werden.
3. Der Code nach der Schleife wird erreicht, da die Exception abgefangen wurde.

```
int[] zahlen = new int[] { 1, 2, 3 };
int summe = 0;
for (int i = 0; i <= zahlen.length; i++) {
    try {
        summe += zahlen[i]; 1

    } catch (ArrayIndexOutOfBoundsException e) {

        System.out.println(e.getMessage()); 2
    }
}
System.out.println("summe: " + summe); 3
```

CATCH BLOCK VERALLGEMEINERN

CATCH BLÖCKE VERALLGEMEINERN

- Catch Blöcke verhalten sich ähnlich zu Methodenargumenten.
- Ein Catch Block für Exception fängt alles, das mit Exception in einer IS-A Beziehung steht.
- Hierbei findet ein impliziter Up-Cast statt.

```
try {
    throw new ArrayIndexOutOfBoundsException("Fehler im Array");
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

EXCEPTIONS WERFEN

EXCEPTIONS WERFEN MIT THROW

- Mit dem Schlüsselwort throw kann jedes Objekt, dass mit Throwable in einer IS-A Beziehung steht, geworfen werden.

```
try {
    throw new ArrayIndexOutOfBoundsException("Fehler im Array");
} catch (ArrayIndexOutOfBoundsException aioobe) {
    System.out.println(aioobe.getMessage());
}
```

EXCEPTIONS WERFEN MIT THROW

- Wird mit throw eine Exception geworfen, erkennt der Compiler, dass der Code danach nicht mehr erreicht werden kann.

```
try {
    throw new ArrayIndexOutOfBoundsException("Fehler im Array");
    System.out.println("Unreachable Code");

} catch (ArrayIndexOutOfBoundsException aioobe) {
    System.out.println(aioobe.getMessage());
}
```

EIGENE EXCEPTIONS

Exceptions erweitern

EIGENE EXCEPTIONS

- Die Exception Klassen können mit einer eigenen Klasse erweitert werden, um eigene Exceptions zu schreiben.
- Jede Exception hat einen Konstruktor, dem eine Zeichenkette als Fehlerinformation übergeben werden kann.

```
2
3  public class MyException extends RuntimeException{
4
5      public MyException(){
6          super("Fehler ist aufgetreten ... ");
7      }
8  }
9
```

EIGENE EXCEPTIONS

- Da MyException mit Throwable in einer IS-A Beziehung steht, kann sie auch mit dem Schlüsselwort throw geworfen werden.

```
try {
    throw new MyException();
} catch (MyException e) {
    System.out.println(e.getMessage());
}
```

MEHRERE CATCH - BLÖCKE

FUNKTIONSWEISE TRY MIT MEHREREN CATCH BLÖCKEN

- In einem Try Block kann es zu verschiedenen Exceptions kommen. Diese können durch mehrere, aufeinander folgende Catch – Blöcke abgefangen werden.

```
String text = "";
int[] zahlen = new int[0];

try {
    char c = text.charAt(1);
    int i = zahlen[1];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
} catch (StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

FUNKTIONSWEISE TRY MIT MEHREREN CATCH BLÖCKEN

- Je nachdem, welche Exception im Try Block auftritt, wird sie vom entsprechenden Catch Block gefangen.

```
String text = "";
int[] zahlen = new int[0];

try {
    char c = text.charAt(1);
    int i = zahlen[1];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
} catch (StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

FUNKTIONSWEISE TRY MIT MEHREREN CATCH BLÖCKEN

- Liegen die Exceptions in einer Vererbungshierarchie, muss bei mehreren Catch Blöcken auf die Reihenfolge geachtet werden.
- Die speziellere Exception muss dabei immer vor der allgemeineren kommen.

```
try {

} catch (StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
} catch (IndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

FUNKTIONSWEISE TRY MIT MEHREREN CATCH BLÖCKEN

- Liegen die Exceptions in einer Vererbungshierarchie, muss bei mehreren Catch Blöcken auf die Reihenfolge geachtet werden.
- Der allgemeinere Catch Block mit IndexOutOfBoundsException würde eine StringIndexOutOfBoundsException auch fangen, daher ist der untere Catch Block nicht erreichbar.

```
try {

} catch (IndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
} catch (StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

MULTI CATCH

MULTI CATCH

- Mehrere Exceptions können durch ein Oder in einem Catch Block zusammengefasst werden.

```
try {

} catch (StringIndexOutOfBoundsException | ArrayIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

- Diese dürfen aber nicht direkt in einer IS-A Beziehung stehen.

```
try {

} catch (StringIndexOutOfBoundsException | RuntimeException e) {
    System.out.println(e.getMessage());
}
```

FINALLY

PAuFa

FUNKTIONSWEISE FINALLY

- Der Finally Block wird zum Schließen von Ressourcen verwendet.
- Der Code im Finally Block wird immer ausgeführt, egal ob ein Fehler auftritt oder nicht.
- Die einzige Ausnahme: System.exit() im Try oder Catch Block

```
Scanner sc = new Scanner(System.in);
try {

} catch (Exception e) {
    System.out.println(e.getMessage());
} finally {
    System.out.println("finally wird immer ausgeführt");
    System.out.println("egal ob ein Fehler auftritt oder nicht");
    System.out.println("finally wird zum Aufräumen verwendet");
    sc.close();
}
```

SYNTAX TRY CATCH FINALLY

SYNTAX TRY CATCH FINALLY

- Ein Try Block darf nicht alleine stehen.
- Als Block immer mit finally, catch oder beidem.

```
try {

} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

```
try {

}finally {

}
```

```
try {

}catch (Exception e) {
    System.out.println(e.getMessage());
}finally {

}
```

THROWS

Deklaration mit throws

DEKLARATION MIT THROWS

- Das Schlüsselwort throws wird verwendet, um an einer Methode den Hinweis anzuhängen, dass in ihr ein Fehler auftreten könnte.

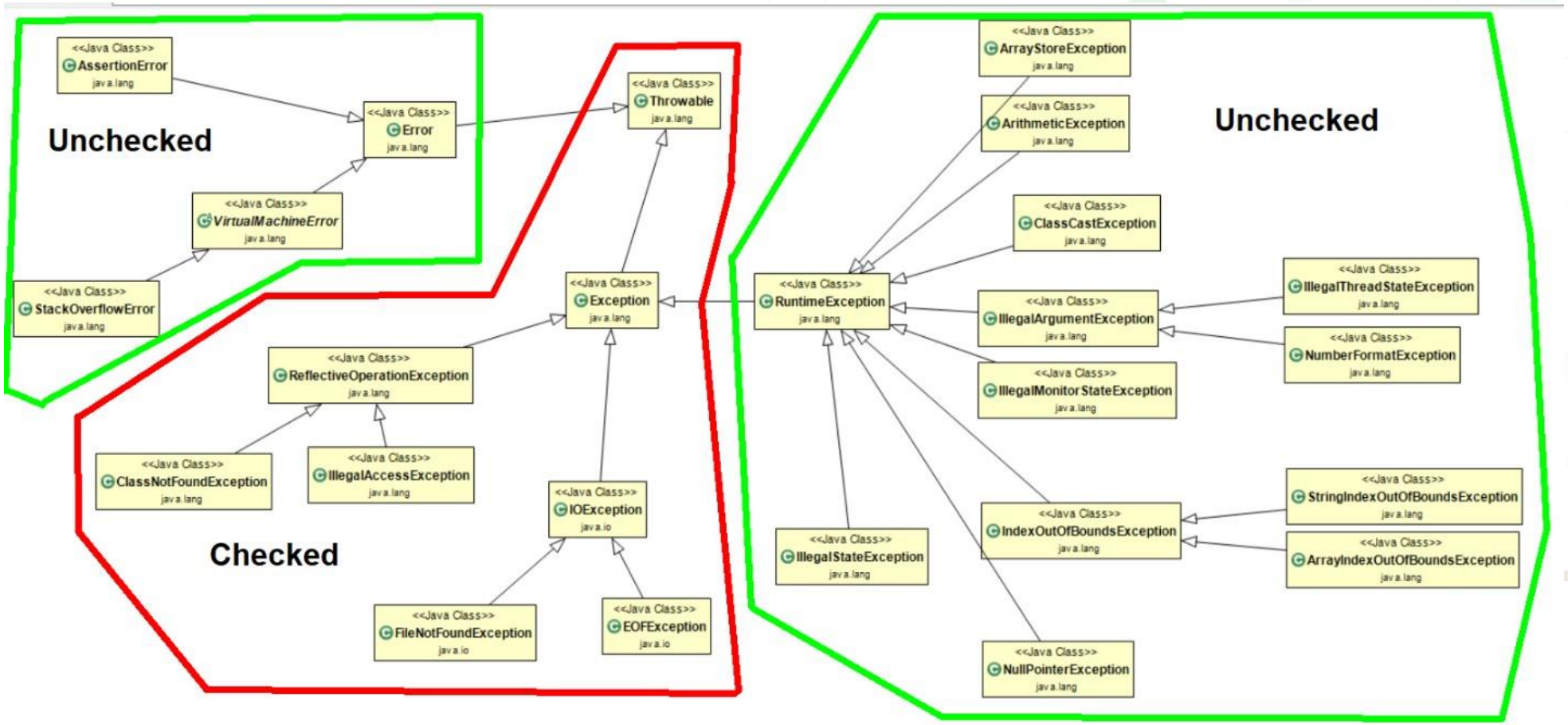
```
public void myMethod() throws Exception{

}
```

ARTEN VON EXCEPTIONS

Checked oder Unchecked

ARTEN VON EXCEPTIONS– CHECKED/UNCHECKED

- Exceptions werden in zwei Gruppen eingeteilt, checked und unchecked.



ARTEN VON EXCEPTIONS–UNCHECKED

- Unchecked Exceptions sind Fehler, die nicht auftreten sollen.
- D.h. es soll kein Code geschrieben werden, mit dem sich das Programm vom Fehlerfall erholt.
- Der Fehler soll im Code beseitigt werden, z.B. durch Kontrollieren der Array Länge.
- Unchecked Exception verlangen keine Fehlerbehandlung und ohne sie geworfen werden

```
throw new RuntimeException();
```

ARTEN VON EXCEPTIONS–CHECKED

- Checked Exceptions sind Fehler, die durch Code nicht verhindert werden können, bzw. nicht vorhersehbar sind.
- So kann z.b. ein Datenstrom durch externen Einfluss unterbrochen werden und eine IOException auslösen.
- Checked Exceptions unterliegen der „Handle or Declare Rule“, d.h sie erzwingen eine Behandlung durch Try – Catch oder eine Deklaration durch throws.

```
//Unhandled exception type Exception
throw new Exception();
```

```
try {
    throw new Exception();
}catch(Exception e) {

}
```

```
public void myMethod() throws Exception{
    throw new Exception();
}
```

ARTEN VON EXCEPTIONS–CHECKED

- Wenn eine Methode eine checked Exception durch throws deklariert, wird die aufrufende Stelle vom Compiler gezwungen, den Fehler zu behandeln oder wieder zu deklarieren.

```
public static void main(String[] args) {
    myMethod();
}

public static void myMethod() throws Exception{
    throw new Exception();
}
```

ARTEN VON EXCEPTIONS–CHECKED

- Dieser Zwang wird durch die Deklaration der Methode ausgelöst, unabhängig davon, ob in der Methode ein Fehler auftritt oder nicht.

```
public static void main(String[] args) {
    myMethod();
}

public static void myMethod() throws Exception{

}
```

ARTEN VON EXCEPTIONS–CHECKED

- Die aufrufende Stelle kann den Fehler entweder durch Try Catch behandeln

```
public static void main(String[] args) {
    try {
        myMethod();
    }catch(Exception e) {

    }
}
public static void myMethod() throws Exception{

}
```

ARTEN VON EXCEPTIONS–CHECKED

- oder auch wieder deklarieren.

```
public static void main(String[] args) throws Exception {

        myMethod();

}
public static void myMethod() throws Exception{

}
```

EXCEPTIONS IN DER VERERBUNG

REGELN FÜR EXCEPTIONS IN DER VERERBUNG

Bei checked Exceptions

Die überschreibende Methode darf nicht:

- Eine allgemeinere Exception deklarieren als die überschriebene Methode.
- Eine neue checked Exception deklarieren.

Die überschreibende Methode darf:

- Die Exception der überschriebenen Methode weglassen.
- Eine speziellere Exception deklarieren als die überschriebene Methode.

REGELN FÜR EXCEPTIONS IN DER VERERBUNG

Bei unchecked Exceptions

Für unchecked Exceptions gelten die Regeln für „new“ oder Verallgemeinern nicht.

Die überschreibende Methode gann beliebig neue unchecked Exceptions hinzufügen.

TRY MIT RESSOURCEN

TRY MIT RESSOURCEN

- In Java 7 wird Try mit Ressoucen eingeführt.
- Try mit Ressourcen vereinfacht das Schließen von Ressourcen, da das Schließen im Finally Block entfällt.

	try-catch-finally	try-with-resources
Resource declared	Before `try` keyword	In parentheses within `try` declaration
Resource initialized	In `try` block	In parentheses within `try` declaration
Resource closed	In `finally` block	Nowhere—happens automatically
Required keywords	`try` One of `catch` or `finally`	`try`

CLOSEABLE UND AUTOCLOSEABLE

CLOSEABLE + AUTOCLOSEABLE

- Damit eine Klasse im Try mit Ressourcen verwendet werden kann, muss sie eines der Interfaces Closeable und Autocloseable implementieren.
- Das Interface Closeable existiert seit Java 5, AutoCloseable seit Java 7.

```
public interface AutoCloseable {
     * Closes this resource, relinqui
    void close() throws Exception;
}
```

```
public interface Closeable extends AutoCloseable {

     * Closes this stream and releases any system r
    public void close() throws IOException;
}
```

CLOSEABLE + AUTOCLOSEABLE

- Die Java Designer wollten bei der close Methode ein paar Dinge ändern, also haben sie AutoCloseable als Eltern Interface erstellt.
- Sie sollte generischer werden durch die Deklaration von Exception statt IOException.
- Die Verwendung im Try mit Ressourcen ist mit beiden Interfaces möglich

	AutoCloseable	Closeable
Extends	None	`AutoCloseable`
`close` method throws	`Exception`	`IOException`
Must be idempotent (can call more than once without side effects)	No, but encouraged	Yes

TRY MIT RESSOURCEN - SYNTAX

TRY MIT RESSOURCEN - SYNTAX

- Beim Try mit Ressourcen werden hinter das Schlüsselwort try in einer runden Klammer die Ressourcen erstellt.

```
try(Reader r = new FileReader(new File("testfile.txt"))){
    //do Reader Stuff
} catch (Exception e) {
    e.printStackTrace();
}
```

- Die Ressourcen müssen vom Typ Closeable oder AutoCloseable sein.
- Folgendes ist nicht erlaubt, da String nicht vom Typ AutoCloseable ist.

```
try (String s = ""){

}
```

TRY MIT RESSOURCEN - SYNTAX

- Anders als beim herkömmlichen Try – Catch kann bei Try Mit Ressourcen der Try Block alleine stehen, wenn die close Methode der Ressource keine Exception deklariert.
- Sonst gelten die gleichen Regeln wie bei Try - Catch

```
try(One o = new One()){

}
```

```
class One implements AutoCloseable{
    @Override
    public void close(){
        System.out.println("Close - One");
    }
}
```

ABLAUF

PAuFa

ABLAUF

- Gegeben sind die beiden Klassen One und Two
- Beide implementieren AutoCloseable

```
class One implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Close - One");
    }
}
class Two implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Close - Two");
    }
}
```

ABLAUF

- Die Verwendung von Try mit Ressourcen ermöglicht zusätzlich zum impliziten Finally Block die Verwendung eines herkömmlichen Finally Blockes

```
public static void main(String[] args) {
    try(One one = new One(); Two two = new Two()){
        System.out.println("Try");
    } catch (Exception e) {

    }finally {
        System.out.println("finally");
    }
}
```

ABLAUF

1. Try Block wird ausgeführt.
2. Die Ressourcen werden in umgekehrter Reihenfolge geschlossen, in der sie erstellt wurden.
3. Der Finally Block wird ausgeführt.

```
Try
Close - Two
Close - One
finally
```

ABLAUF

- Kommt es im Try Block zu einer Exception, werden erst die Ressourcen geschlossen, dann der Catch Block ausgeführt und am Ende der Finally Block.

```
public static void main(String[] args) {
    try(One one = new One(); Two two = new Two()){
        System.out.println("Try");
        throw new RuntimeException();
    } catch (Exception e) {
        System.out.println("Fehler aufgetreten");
    }finally {
        System.out.println("finally");
    }
}
```

```
Try
Close - Two
Close - One
Fehler aufgetreten
finally
```

IOEXCEPTIONS

IOEXCEPTIONS

- Bei IO Exceptions ist darauf zu achten, dass sie sich im Paket java.io befinden, nicht im Paket java.lang. Es ist also ein Import oder die Angabe des voll qualifizierten Klassennamens nötig.

```
//IOException cannot be resolved to a type
throw new IOException();

throw new java.io.IOException();
```