

14. Ausnahmebehandlung

14.1 Prinzip

14.2 Try-Anweisung

14.3 Arten von Ausnahmen

14.4 Implementierung von Ausnahmen

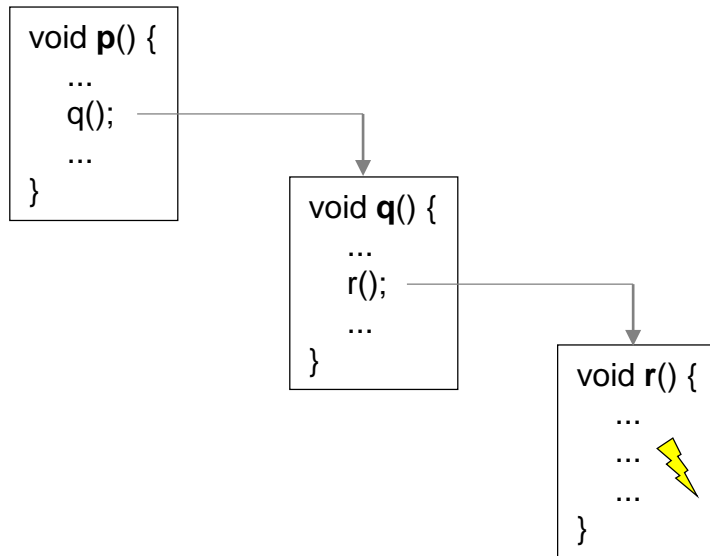
14.5 Suche nach passender Catch-Klausel

14.6 Spezifikation von Ausnahmen im Methodenkopf

Motivation



Fehler können nicht immer dort behandelt werden, wo sie auftreten



Annahme: irgendeine Operation kann hier nicht ausgeführt werden. Wie soll $r()$ reagieren?

- Fehlermeldung ausgeben?
- Programm abbrechen?
- einfach weiterlaufen?
- Korrekturmaßnahmen?
- ...

Vielleicht möchte $q()$ oder $r()$ reagieren?

Lösung

$r()$ muss den Fehler an $q()$ melden,
 $q()$ muss ihn an $p()$ melden,
bis ihn jemand behandelt

Fehlerbehandlung in früheren Zeiten



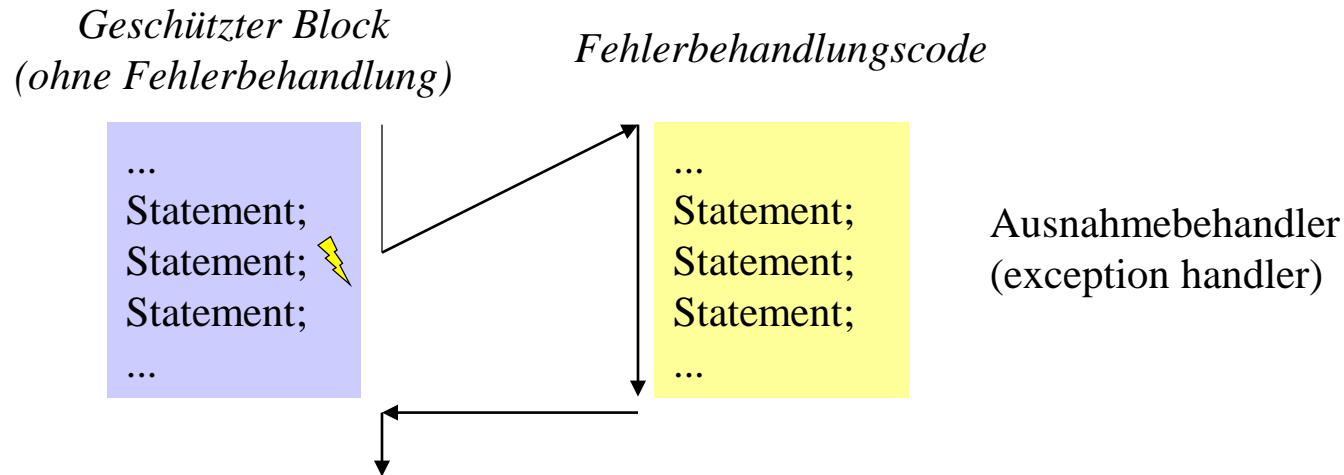
Jede Methode lieferte einen Fehlercode

```
static final int OK = 0;
int result;
result = p(...);
if (result == OK) {
    result = q(...);
    if (result == OK) {
        result = r(...);
        if (result == OK) {
            ...
        } else error(...);
    } else error(...);
} else error(...);
```

Probleme

- aufwändig: vor lauter Fehlerbehandlung sieht man eigentliches Programm nicht mehr
- Abfragen des Fehlercodes kann vergessen werden
- Abfragen des Fehlercodes wird oft aus Bequemlichkeit nicht durchgeführt
- Funktionen können neben Fehlercode kein anderes Ergebnis mehr liefern

Idee der Ausnahmebehandlung in Java



Wenn im geschützten Block ein Fehler (eine Ausnahme) auftritt:

- Ausführung des geschützten Blocks wird abgebrochen
- Fehlerbehandlungscode wird ausgeführt
- Programm setzt nach dem geschützten Block fort

14. Ausnahmebehandlung

14.1 Prinzip

14.2 Try-Anweisung

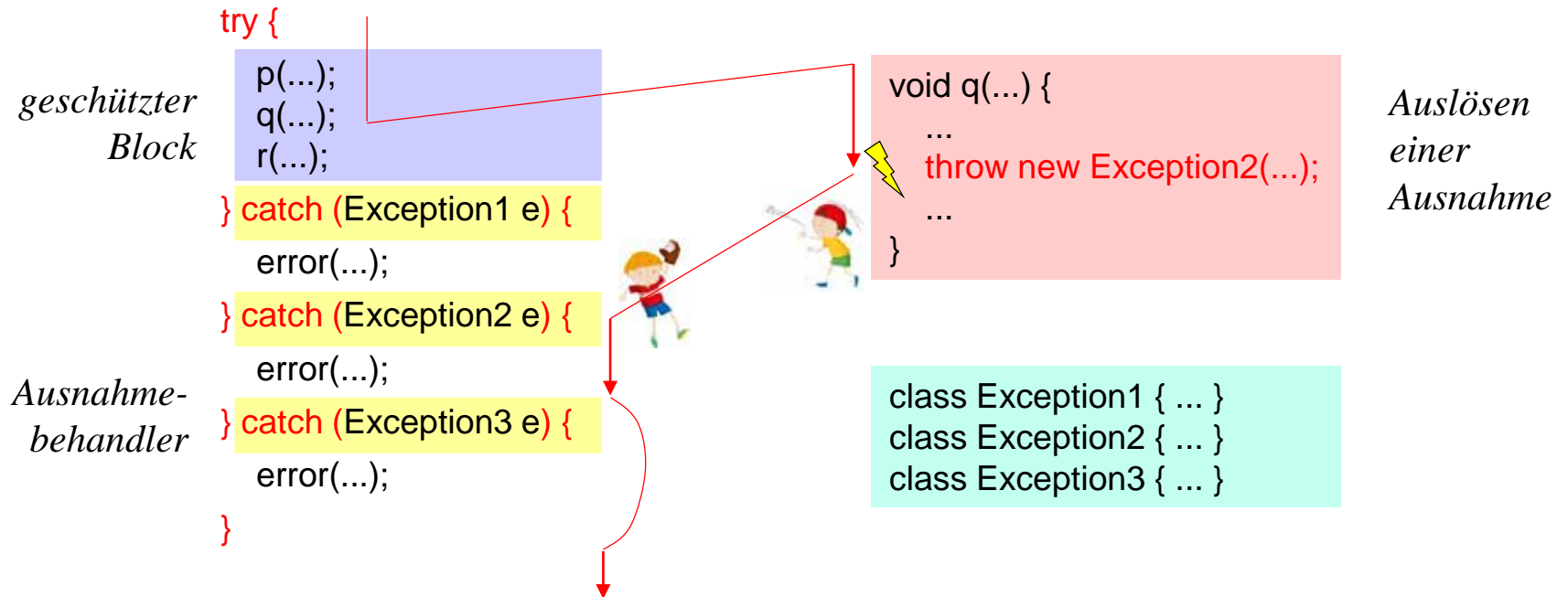
14.3 Arten von Ausnahmen

14.4 Implementierung von Ausnahmen

14.5 Suche nach passender Catch-Klausel

14.6 Spezifikation von Ausnahmen im Methodenkopf

Try-Anweisung in Java



Vorteile

- Fehlerfreier Fall und Fehlerfälle sind sauberer getrennt
- Man kann nicht vergessen, einen Fehler zu behandeln
(Compiler prüft, ob es zu jeder möglichen Ausnahme einen Behandler gibt)

14. Ausnahmebehandlung

14.1 Prinzip

14.2 Try-Anweisung

14.3 Arten von Ausnahmen

14.4 Implementierung von Ausnahmen

14.5 Suche nach passender Catch-Klausel

14.6 Spezifikation von Ausnahmen im Methodenkopf

Arten von Ausnahmen



Laufzeitfehler (*Runtime Exceptions*)

werden von der Java-VM ausgelöst (auf Grund illegaler Operationen)

- Division durch 0 *ArithmeticException*
- Zugriff über null-Zeiger *NullPointerException*
- Indexüberschreitung *ArrayIndexOutOfBoundsException*

Müssen nicht behandelt werden

Wenn sie nicht behandelt werden, stürzt das Programm mit einer Fehlermeldung ab

Geprüfte Ausnahmen (*Checked Exceptions*)

werden vom Benutzercode ausgelöst (*throw*-Anweisung)

- vordefinierte Ausnahmen z.B. *FileNotFoundException*
- selbst definierte Ausnahmen z.B. *MyException*

Müssen behandelt werden.

Compiler prüft, ob sie abgefangen werden

Hierarchie der Ausnahmeklassen



Throwable

Error

LinkageError

VirtualMachineError

...

Exception

RuntimeException

NullPointerException

ArithmeticException

IndexOutOfBoundsException

...

IOException

FileNotFoundException

...

AWTException

...

MyException1

MyException2

...

Fehler, die nicht am Programm liegen
(müssen nicht abgefangen werden)

von der VM ausgelöste Fehler
(müssen nicht abgefangen werden)

vordefinierte Ausnahmen
(müssen abgefangen werden)

selbst definierte Ausnahmen
(müssen abgefangen werden)

14. Ausnahmebehandlung

14.1 Prinzip

14.2 Try-Anweisung

14.3 Arten von Ausnahmen

14.4 Implementierung von Ausnahmen

14.5 Suche nach passender Catch-Klausel

14.6 Spezifikation von Ausnahmen im Methodenkopf

Ausnahmen sind als Klassen codiert



Fehlerinformationen stehen in einem Ausnahme-Objekt

```
class Exception extends Throwable {  
    private String msg;           // Fehlermeldung  
    Exception(String msg) {...}   // erzeugt neues Ausnahmeobjekt mit Fehlermeldung  
    String getMessage() {...}    // liefert gespeicherte Fehlermeldung  
    String toString() {...}      // liefert Art der Ausnahme und gespeichert Fehlermeldung  
    void printStackTrace() {...} // gibt Methodenaufkette aus  
    ...  
}
```

```
java.io.FileNotFoundException: Sample.txt  
(The system cannot find the file specified)
```

```
java.io.FileNotFoundException: Sample.txt  
(The system cannot find the file specified)  
    at java.io.FileInputStream.open0(Native Method)  
    at java.io.FileInputStream.open(Unknown Source)  
    at java.io.FileInputStream.<init>(Unknown Source)  
    at java.io.FileInputStream.<init>(Unknown Source)  
    at Test.bar(test.java:33)  
    at Test.foo(test.java:50)  
    at Test.main(test.java:56)
```

Eigene Ausnameklasse (speichert Informationen über speziellen Fehler)

```
class MyException extends Exception {  
    private int errorCode;  
    MyException(String msg, int errorCode) { super(msg); this.errorCode = errorCode; }  
    int getErrorCode() {...}  
    // toString(), printStackTrace(), ... von Exception geerbt  
}
```

Ausnahmen sind als Klassen codiert



Fehlerinformationen stehen in einem Ausnahme-Objekt

```
class Exception extends Throwable {  
    Exception(String msg) {...}        // erzeugt neues Ausnahmeobjekt mit Fehlermeldung  
    String getMessage() {...}         // liefert gespeicherte Fehlermeldung  
    String toString() {...}           // liefert Art der Ausnahme und gespeicherte Fehlermeldung  
    void printStackTrace() {...}       // gibt Methodenaufruflkette aus  
    ...  
}
```

Eigene Ausnameklasse (speichert Informationen über speziellen Fehler)

```
class MyException extends Exception {  
    private int errorCode;  
    MyException(String msg, int errorCode) { super(msg); this.errorCode = errorCode; }  
    int getErrorCode() {...}  
    // toString(), printStackTrace(), ... von Exception geerbt  
}
```

Throw-Anweisung

Löst eine Ausnahme aus

```
throw new MyException("invalid operation", 17);
```

"Wirft" ein Ausnahmeobjekt mit entsprechenden Fehlerinformationen

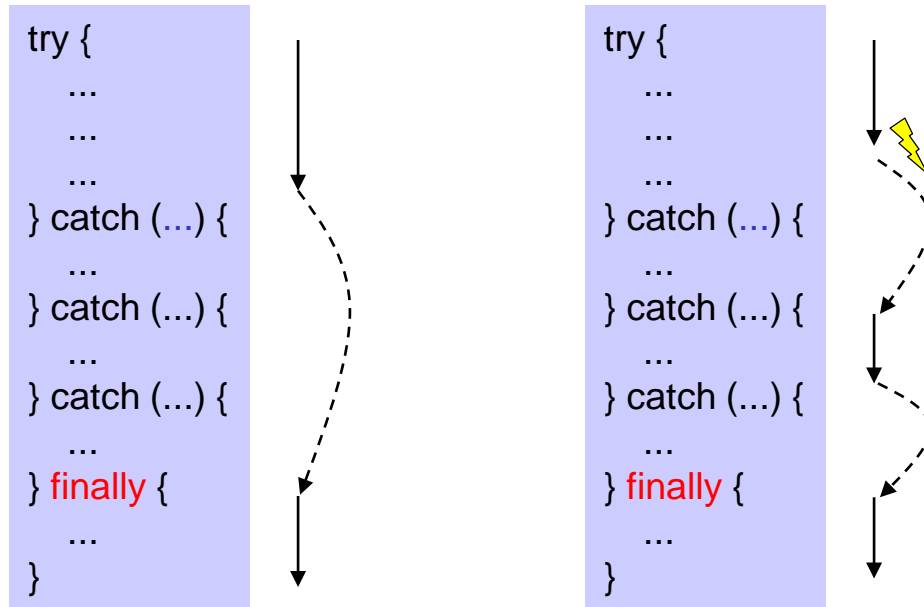
- bricht normale Programmausführung ab
- sucht passenden Ausnahmebehandler (Catch-Block)
- führt Ausnahmebehandler aus und übergibt ihm Ausnahmeobjekt als Parameter
- setzt nach der Try-Anweisung fort, zu der der Catch-Block gehört

Catch-Blöcke

```
try {
    ...
} catch (MyException e) {
    Out.println(e.getMessage() + ", error code = ", + e.getErrorCode());
} catch (NullPointerException e) {
    ...
} catch (Exception e) {
    ...
}
```

- Beliebig viele
- Passender Catch-Block wird an Hand des Ausnahme-Typs ausgewählt
- Catch-Blöcke werden sequentiell abgesucht
Achtung: speziellere Ausnahme-Typen müssen vor allgemeineren stehen

Finally-Block




- Ist optional
- Wird am Ende der Try-Anweisung immer ausgeführt
egal, ob im geschützten Block ein Fehler auftrat oder nicht

Zweck des Finally-Blocks




Zum Sicherstellen, dass Abschlussarbeiten auch im Fehlerfall ausgeführt werden

```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
    In.close();  
} catch (...) {  
    ...  
}
```

falsch

Datei wird im Fehlerfall nicht geschlossen

```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
} catch (...) {  
    ...  
} finally {  
    In.close();  
}
```

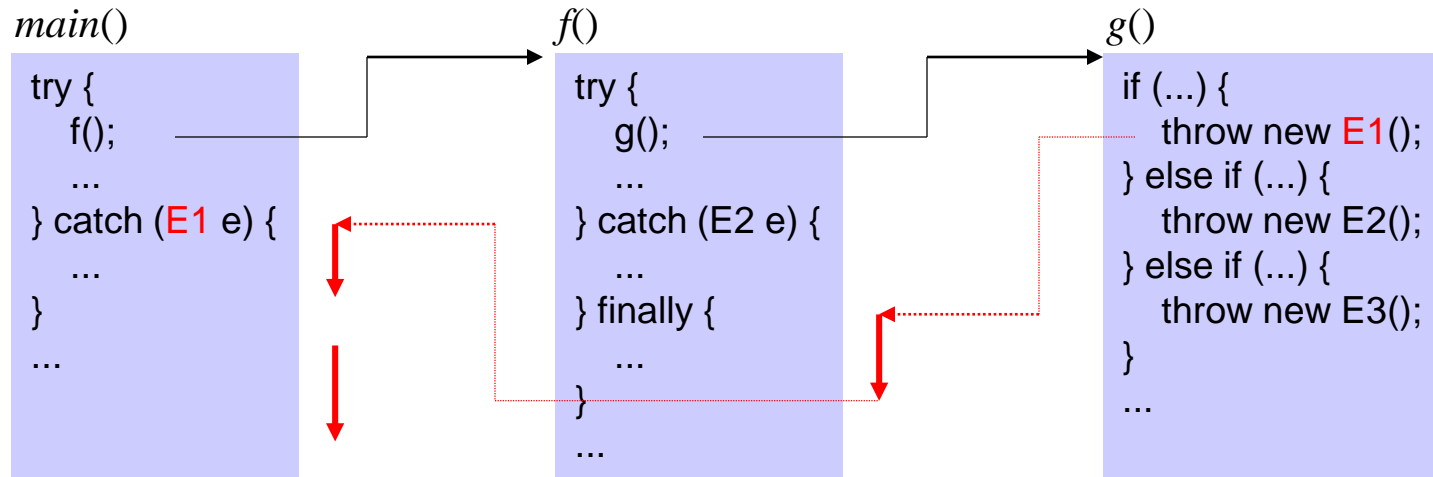
richtig

Datei wird auf jeden Fall geschlossen

14. Ausnahmebehandlung

- 14.1 Prinzip
- 14.2 Try-Anweisung
- 14.3 Arten von Ausnahmen
- 14.4 Implementierung von Ausnahmen
- 14.5 Suche nach passender Catch-Klausel
- 14.6 Spezifikation von Ausnahmen im Methodenkopf

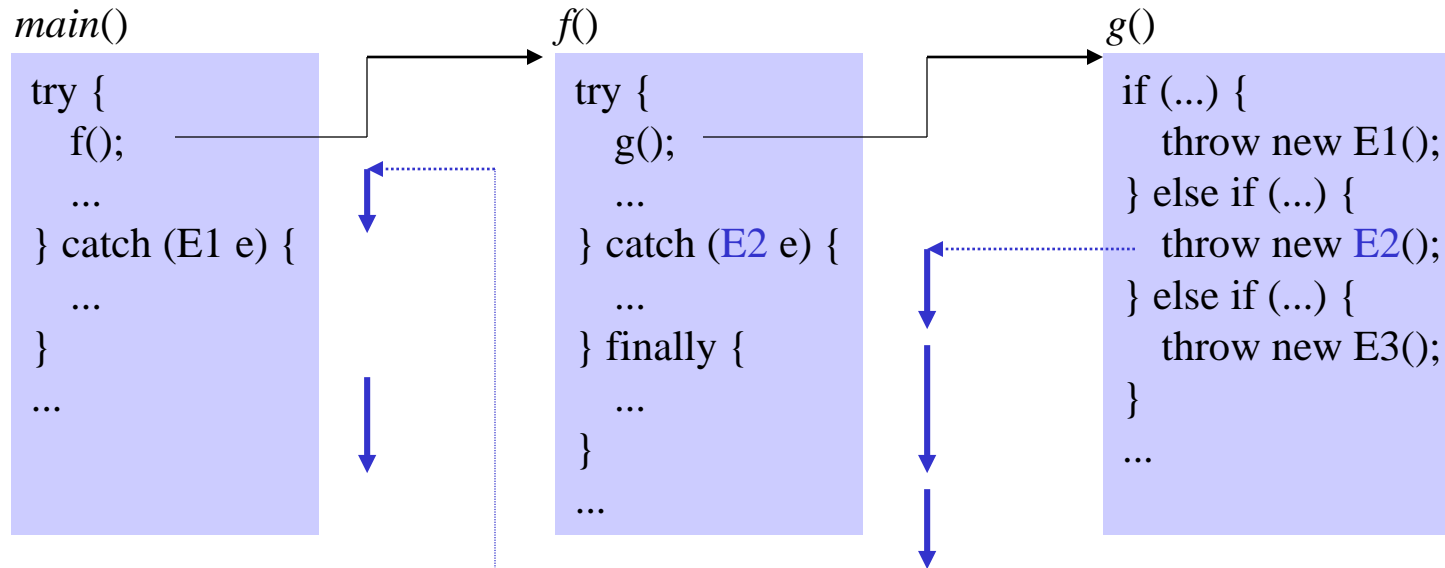
Suche der passenden Catch-Klausel



`throw new E1();`

- keine Try-Anweisung in `g()` \Rightarrow bricht `g()` ab
- kein passender Catch-Block in `f()` \Rightarrow führt Finally-Block in `f()` aus und bricht `f()` dann ab
- führt Catch-Block für `E1` in `main()` aus
- setzt nach Try-Anweisung in `main()` fort

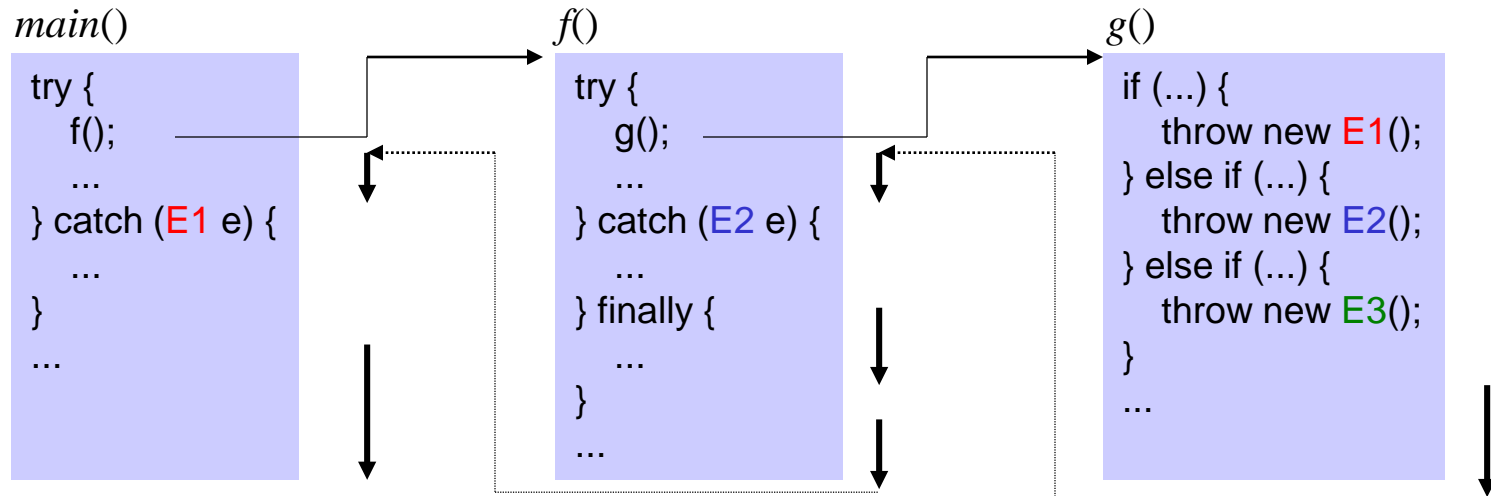
Suche der passenden Catch-Klausel



`throw new E2();`

- keine Try-Anweisung in `g()` \Rightarrow bricht `g()` ab
- führt Catch-Block für `E2` in `f()` aus
- führt Finally-Block in `f()` aus
- setzt nach Try-Anweisung in `f()` fort

Suche der passenden Catch-Klausel



throw new E3();

- Compiler meldet einen Fehler, weil *E3* nirgendwo in der Ruferkette abgefangen wird

Fehlerfreier Fall

- führt *g()* zu Ende aus
- führt Try-Block in *f()* zu Ende aus
- führt Finally-Block in *f()* aus
- setzt nach Finally-Block in *f()* fort

14. Ausnahmebehandlung

14.1 Prinzip

14.2 Try-Anweisung

14.3 Arten von Ausnahmen

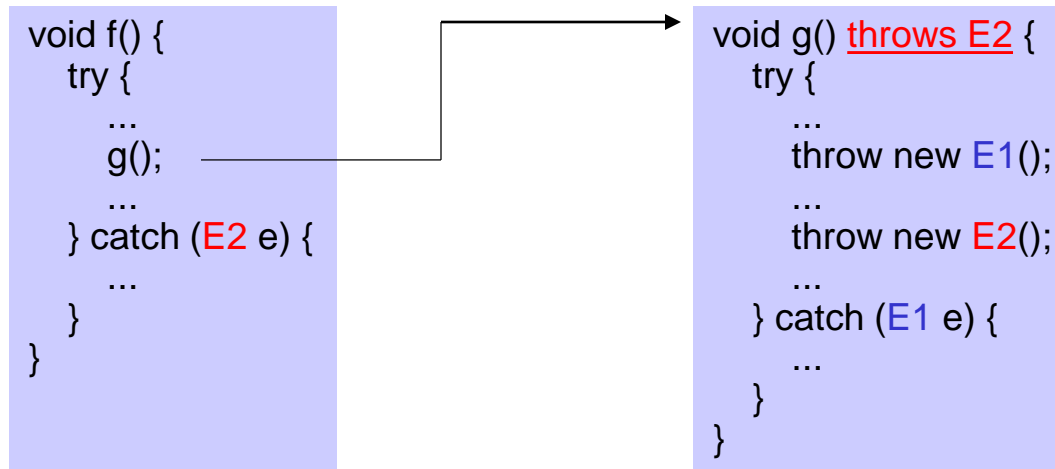
14.4 Implementierung von Ausnahmen

14.5 Suche nach passender Catch-Klausel

14.6 Spezifikation von Ausnahmen im Methodenkopf

Spezifikation von Ausnahmen im Methodenkopf

Wenn eine Methode eine Ausnahme an den Rufer weiterleitet, muss sie das in ihrem Methodenkopf mit einer Throws-Klausel spezifizieren



Compiler weiß dadurch, dass `g()` eine `E2`-Ausnahme auslösen kann.

Wer `g()` aufruft, muss daher

- entweder `E2` abfangen
- oder `E2` im eigenen Methodenkopf mit einer *Throws-Klausel* spezifizieren

▷ **Man kann nicht vergessen, eine Ausnahme zu behandeln!**

Eigentlich müsste es heißen



void main(...) throws E3

```
try {  
    f();  
    ...  
} catch (E1 e) {  
    ...  
}  
...
```

void f() throws E1, E3

```
try {  
    g();  
    ...  
} catch (E2 e) {  
    ...  
} finally {  
    ...  
}  
...
```

void g() throws E1, E2, E3

```
if (...) {  
    throw new E1();  
} else if (...) {  
    throw new E2();  
} else if (...) {  
    throw new E3();  
}  
...
```

Auch *main()* kann eine Throws-Klausel haben