8 Fehlerobjekte: Werfen, Fangen, Behandeln

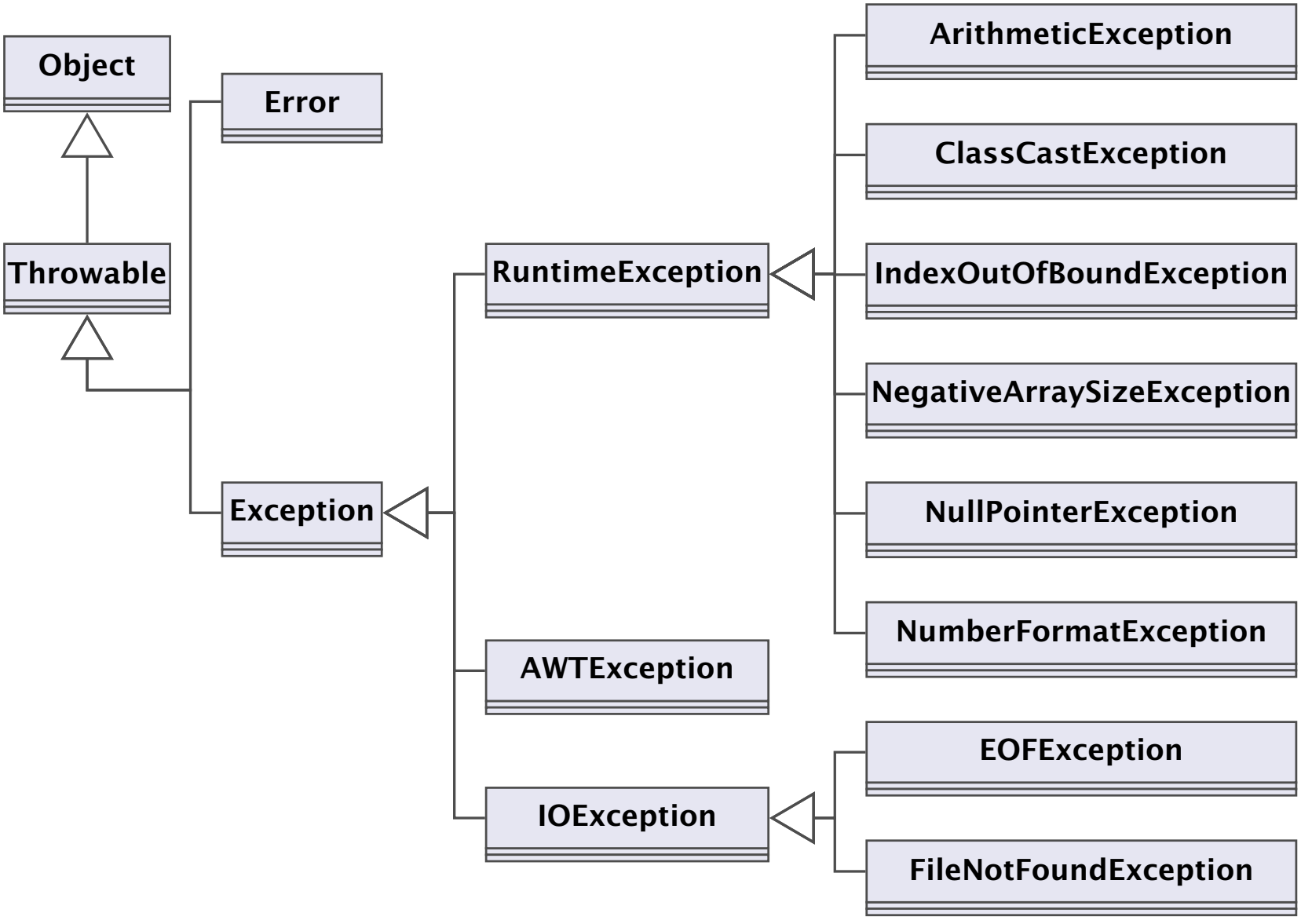
- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehlerobjekt kann gefangen und geeignet behandelt werden.

Idee

Trennung von

- normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken,. . .)

Fehlerklassen



Fehlerklassen

Die direkten Unterklassen von `Throwable` sind:

- `Error` — für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` — für bewältigbare Fehler oder Ausnahmen.

unchecked exception

Ausnahmen der Klasse `Error` und `RuntimeException` müssen nicht im Methodenkopf deklariert werden.

checked exception

Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden!!!

Fehlerklassen

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten...
- Sie kann, muss aber nicht abgefangen werden.

Arten der Fehlerbehandlung:

- Ignorieren;
- Abfangen und Behandeln dort, wo sie entstehen;
- Abfangen und Behandeln an einer anderen Stelle.

Fehlerbehandlung

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programmausführung ab.

Beispiel:

```
1 public class Zero {
2     public static void main(String[]
          args) {
3         int x = 10;
4         int y = 0;
5         System.out.println(x/y);
6     } // end of main()
7 } // end of class Zero
```

Fehlermeldung

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: „/ by zero“.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Call-Stack.

Fehlerbehandlung

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: `NumberFormatException`

```
1 public class Adding extends MiniJava {
2     public static void main(String[] args) {
3         int x = getInt("1. Zahl:\t");
4         int y = getInt("2. Zahl:\t");
5         write("Summe:\t\t" + (x+y));
6     } // end of main()
7     public static int getInt(String str) {
8 //continued...
```

Erläuterungen

- Das Programm liest zwei `int`-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - . . . weil die Eingabe keine syntaktisch korrekte Zahl ist;
  - . . . weil sonstige unvorhersehbare Ereignisse eintreffen.
- Die Behandlung dieser Fehler ist in der Funktion `getInt()` verborgen. . .

Fehlerbehandlung

```
 9          String s;
10          while (true) {
11              try {
12                  s = readString(str);
13                  return Integer.parseInt(s);
14              } catch (NumberFormatException e) {
15                  System.out.println(
16                       "Falsche Eingabe! ...");
17              } catch (IOException e) {
18                  System.out.println(
19                       "Eingabeproblem: Ende ...");
20                  System.exit(0);
21              }
22          } // end of while
23      } // end of getInt()
24  } // end of class Adding
```

Beispielablauf

```
> java Adding
1. Zahl: abc
Falsche Eingabe! ...
1. Zahl: 0.3
Falsche Eingabe! ...
1. Zahl: 17
2. Zahl: 25
Summe: 42
```

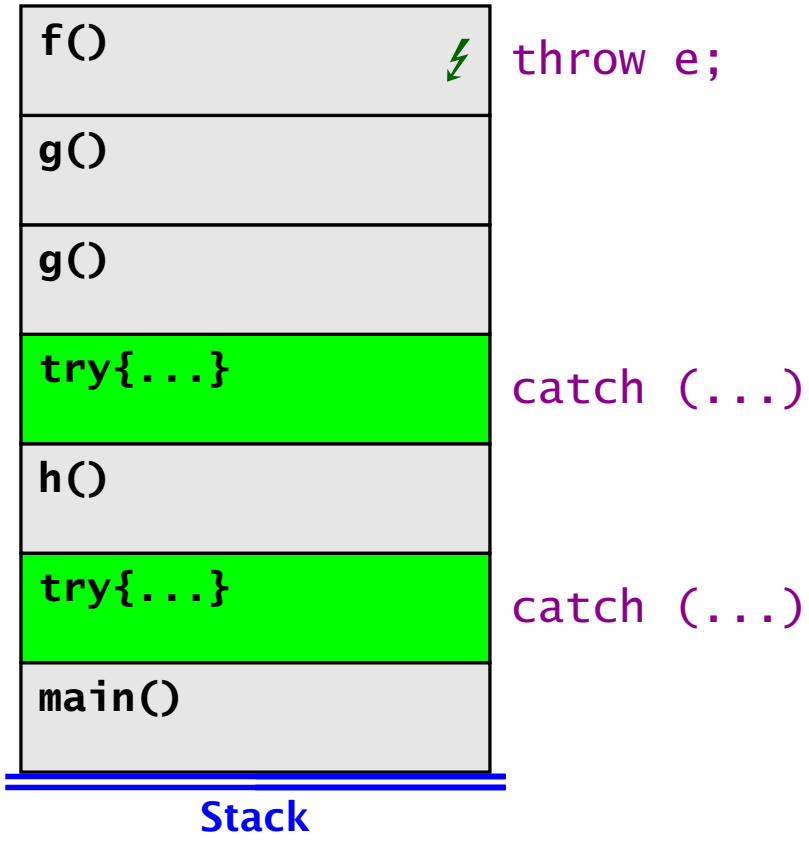
Ausnahmebehandlung

- Ein Exception-Handler besteht aus einem `try`-Block `try{`*`ss`*`}`, in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge *`ss`* kein Fehlerobjekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei `Exc` eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von `Exc` stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

Was passiert auf dem Stack?



Ausnahmebehandlung

- Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

Ausnahmebehandlung

- Die Folge `ss` von Statements wird auf jeden Fall ausgeführt.
- Wird kein Fehler im `try`-Block geworfen, wird sie im Anschluss an den `try`-Block ausgeführt.
- Wird ein Fehler geworfen und mit einer `catch`-Regel behandelt, wird sie nach dem Block der `catch`-Regel ausgeführt.
- Wird der Fehler von keiner `catch`-Regel behandelt, wird `ss` ausgeführt, und dann der Fehler weitergereicht.

Beispiel NullPointerException

```
1  public class Kill {
2      public static void kill() {
3          Object x = null; x.hashCode ();
4      }
5      public static void main(String[] args) {
6          try { kill();
7          } catch (ClassCastException b) {
8              System.out.println("Falsche Klasse!!!");
9          } finally {
10             System.out.println("Leider nix gefangen
                   ...");
11         }
12     } // end of main()
13 } // end of class Kill
```

Resultat:

```
> java Kill
Leider nix gefangen ...
Exception in thread "main" java.lang.NullPointerException
        at Kill.kill(Compiled Code)
        at Kill.main(Compiled Code)
```

Selbstdefinierte Fehler

Exceptions können auch

- selbst definiert und
- selbst geworfen werden.

Beispiel:

```
1 public class Killed extends Exception {
2     Killed() {}
3     Killed(String s) {super(s);}
4 } // end of class Killed
5 public class Kill {
6     public static void kill() throws Killed {
7         throw new Killed();
8     }
9 // continued...
```

Beispiel

```
10      public static void main(String[] args) {
11          try {
12              kill();
13          } catch (RuntimeException r) {
14              System.out.println("RunTimeException "+ r);
15          } catch (Killed b) {
16              System.out.println("Killed It!");
17              System.out.println(b);
18              System.out.println(b.getMessage());
19          }
20      } // end of main
21  } // end of class Kill
```

Selbstdefinierte Fehler

- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden!
- Die Klasse `Exception` verfügt über die Konstruktoren
  `public Exception();`
  `public Exception(String str);`
  (`str` ist die evt. auszugebende Fehlermeldung).
- `throw exc` löst den Fehler `exc` aus — sofern sich der Ausdruck `exc` zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen.
- Ausgabe:

```
Killed It!
Killed
Null
```

Fazit

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Checked Exceptions

Checked Exceptions sind eine Spezialität von Java. In anderen Programmiersprachen wie C#, C++ findet man diese nicht.

Vorteile:

- Man sieht sofort welche Exceptions eine Funktion werfen kann.
- Der Compiler „überprüft“ ob man die Exceptions behandelt.

Mögliche Nachteile:

- Skalierbarkeit: Wenn man viele API-Funktionen aufruft muß man eventuell sehr viele Exceptions angeben, die man werfen könnte.
- Exceptions können Details der Implementierung verraten (e.g. `SQLException`). D.h. es könnte schwierig sein die Implementierung einer API-Funktion später zu ändern.

Exceptions – Hinweise

- Exceptions fangen ist teuer. D.h. man sollte Exceptions nur für außergewöhnliche Situationen nutzen. Nicht zur Ablaufkontrolle.
- Fehler sollten dort behandelt werden, wo sie auftreten.
- Es ist besser spezifischere Fehler zu fangen als allgemeine — z.B. mit `catch(Exception e) {}`
- Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Eine API-Funktion sollte Exceptions werfen, die der Abstraktion der Funktion angemessen sind. (Also `getUser()` sollte z.B. `UserNotFoundException` werfen, anstatt `SQLException`.)
- Programmierfehler (z.B. falsche Verwendung einer Klasse) sollten durch `RunTimeException` signalisiert werden.
- Checked Exceptions sollte man nur verwenden, wenn der Aufrufer sie sinnvoll behandeln kann.

Exception Safety

Es gibt verschiedene Garantien, die eine Objektmethode bzgl. Exceptions erfüllen kann:

Basic Guarantee
Nach einer Exception erfüllt das Objekt alle seine Invarianten. Keine Leaks.

Häufig nicht sehr hilfreich. Das Objekt kann sich beliebig verändert haben; das Gute ist, dass wir das Objekt noch löschen können.

Strong Guarantee
Im Fall einer Exception hat sich das Objekt nicht verändert. (Transaktionales Verhalten; entweder funktioniert alles, oder keine Änderung wird durchgeführt).

No-throw Guarantee
Die Funktion wirft keine Exceptions.