Tritt ein Fehler auf und wird nicht behandelt, bricht die Programm-Ausführung ab.

Beispiel:

```
public class Zero {
   public static main(String[] args) {
      int x = 10;
      int y = 0;
      System.out.println(x/y);
   }  // end of main()
}     // end of class Zero
```

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 Rekursions-Stack.

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

Beispiel: `NumberFormatException`

```
import java.io.*;
public class Adding {
   private static BufferedReader stdin = new BufferedReader
           (new InputStreamReader(System.in));
   public static void main(String[] args) {
      int x = getInt("1. Zahl:\t");
      int y = getInt("2. Zahl:\t");
      System.out.println("Summe:\t\t"+ (x+y));
   }  // end of main()
   public static int getInt(String str) {
      ...
```

- Das Programm liest zwei `int`-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - ... weil keine syntaktisch korrekte Zahl eingegeben wird;
  - ... weil sonstige unvorhersehbare Ereignisse eintreffen :-)
- Die Behandlung dieser Fehler ist in der Funktion `getInt()` verborgen.
- Die `boolean`-Variable der Funktion `getInt()` soll den Wert `true` enthalten, wenn die Eingabe einer Zahl erfolgreich abgeschlossen ist ...

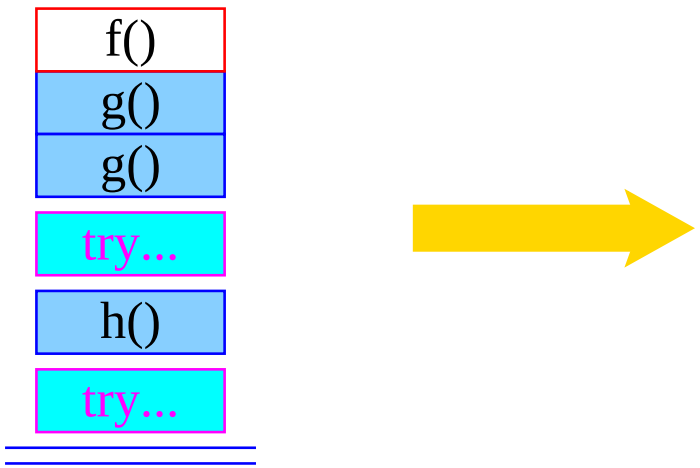
```
        while (true) {
            System.out.print(str);
            System.out.flush();
            try {
                return Integer.parseInt(stdin.readLine());
            } catch (NumberFormatException e) {
                System.out.println("Falsche Eingabe! ...");
            } catch (IOException e) {
                System.out.println("Eingabeproblem: Ende ...");
                System.exit(0);
            }
        }  // end of while
    }      // end of getInt()
}          // end of class Adding
```

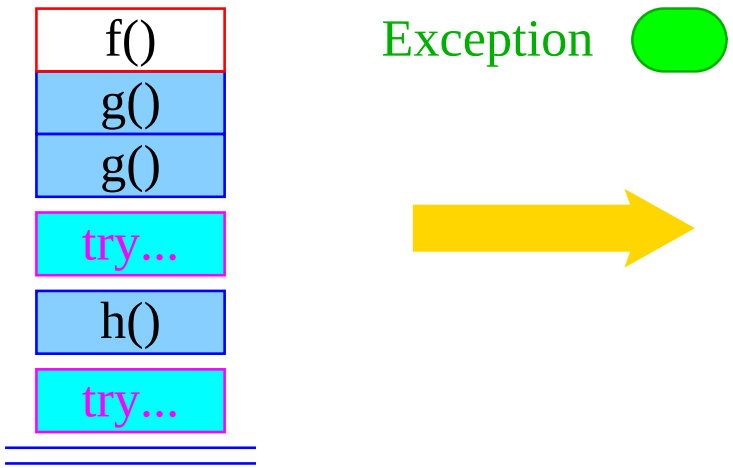
... ermöglicht folgenden Dialog:

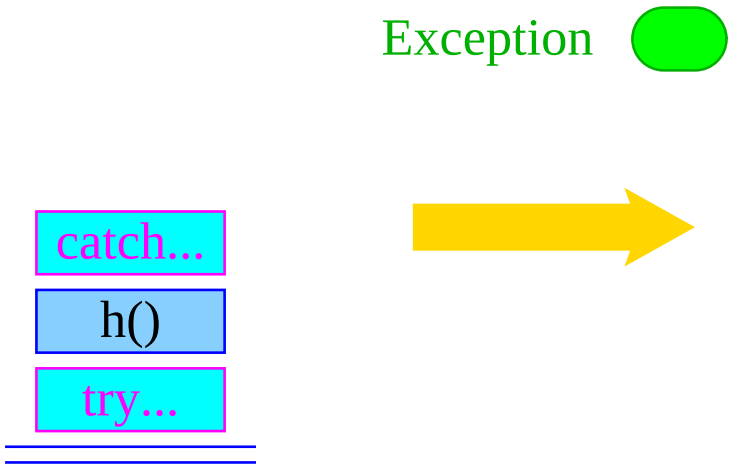
```
> java Adding
1. Zahl:          abc
Falsche Eingabe! ...
1. Zahl:          0.3
Falsche Eingabe! ...
1. Zahl:          17
2. Zahl:          25
Summe:            42
```

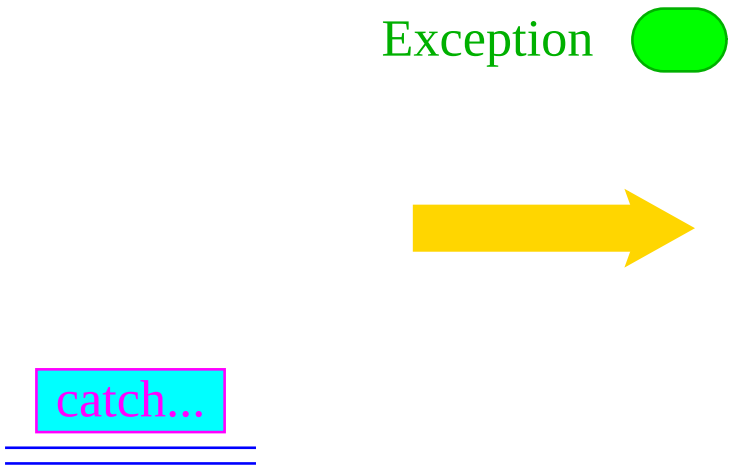
- Ein Exception-Handler besteht aus einem `try`-Block `try {` ss `}` , in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.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

- 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 Fehler-Objekt 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.


f()
g()
g()
try...
h()
try...


f()
g()
g()
try...
h()
try...
Exception


Exception
catch...
h()
try...


Exception
catch...

- 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 IO-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

- 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 IO-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

Achtung:

- 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`

```
public class Kill {
   public static void kill() {
      Object x = null; x.hashCode ();
   }
   public static void main(String[] args) {
      try { kill();
      } catch (ClassCastException b) {
         System.out.println("Falsche Klasse!!!");
      } finally {
         System.out.println("Leider nix gefangen ...");
      }
   }  // end of main()
}     // end of class Kill
```

... liefert:

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

Exceptions können auch

- selbst definiert und
- selbst geworfen werden.

Beispiel:

```
public class Killed extends Exception {
   Killed() {}
   Killed(String s) {super(s);}
} // end of class Killed
public class Kill {
   public static void kill() throws Killed {
      throw new Killed();
   }
   ...
```

```
    public static void main(String[] args) {
       try {
          kill();
       } catch (RuntimeException r) {
          System.out.println("RunTimeException "+ r +"\n");
       } catch (Killed b) {
          System.out.println("Killed It!");
          System.out.println(b);
          System.out.println(b.getMessage());
       }
    } // end of main
}    // end of class Kill
```

- 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.
- Die vordefinierten Fehlerarten reichen oft aus.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Warnung:

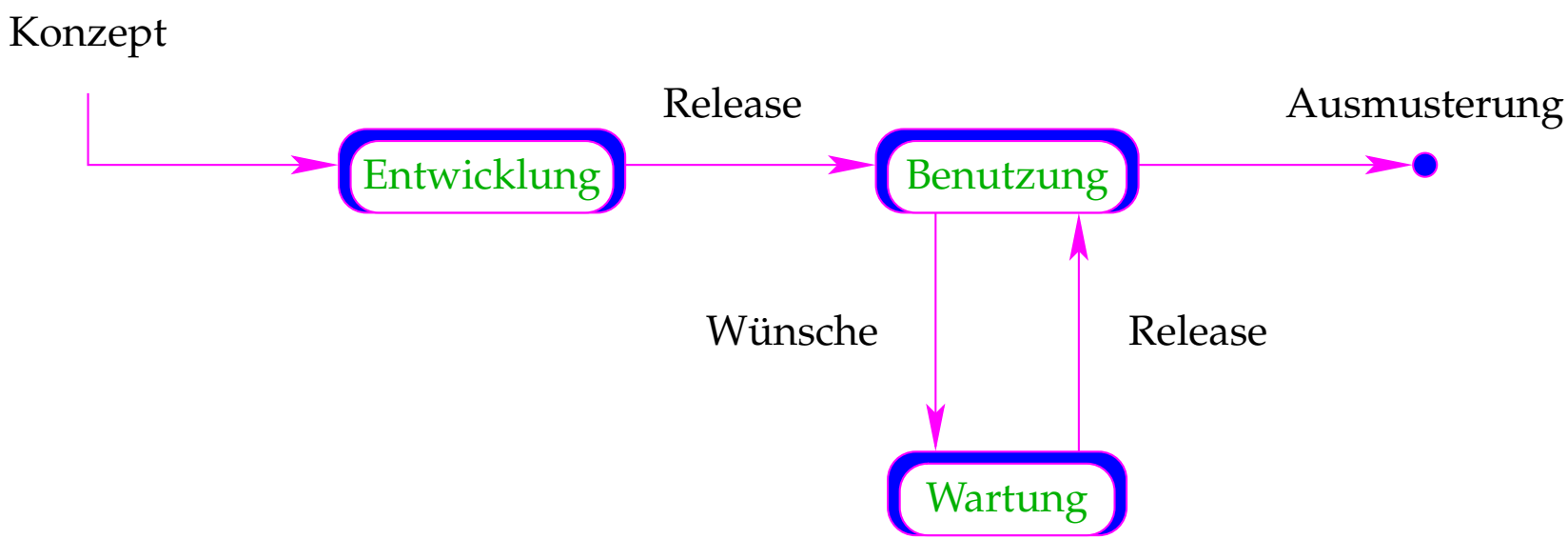
- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Fehler sollten dort behandelt werden, wo sie auftreten :-)
- Es ist besser spezifischere Fehler zu fangen als allgemeine – z.B. mit `catch (Exception e) { }`

18 Programmieren im Großen

Neu:

- Das Programm ist groß.
- Das Programm ist unübersichtlich.
- Das Programm ist teuer.
- Das Programm wird lange benutzt.

Software-Zyklus:

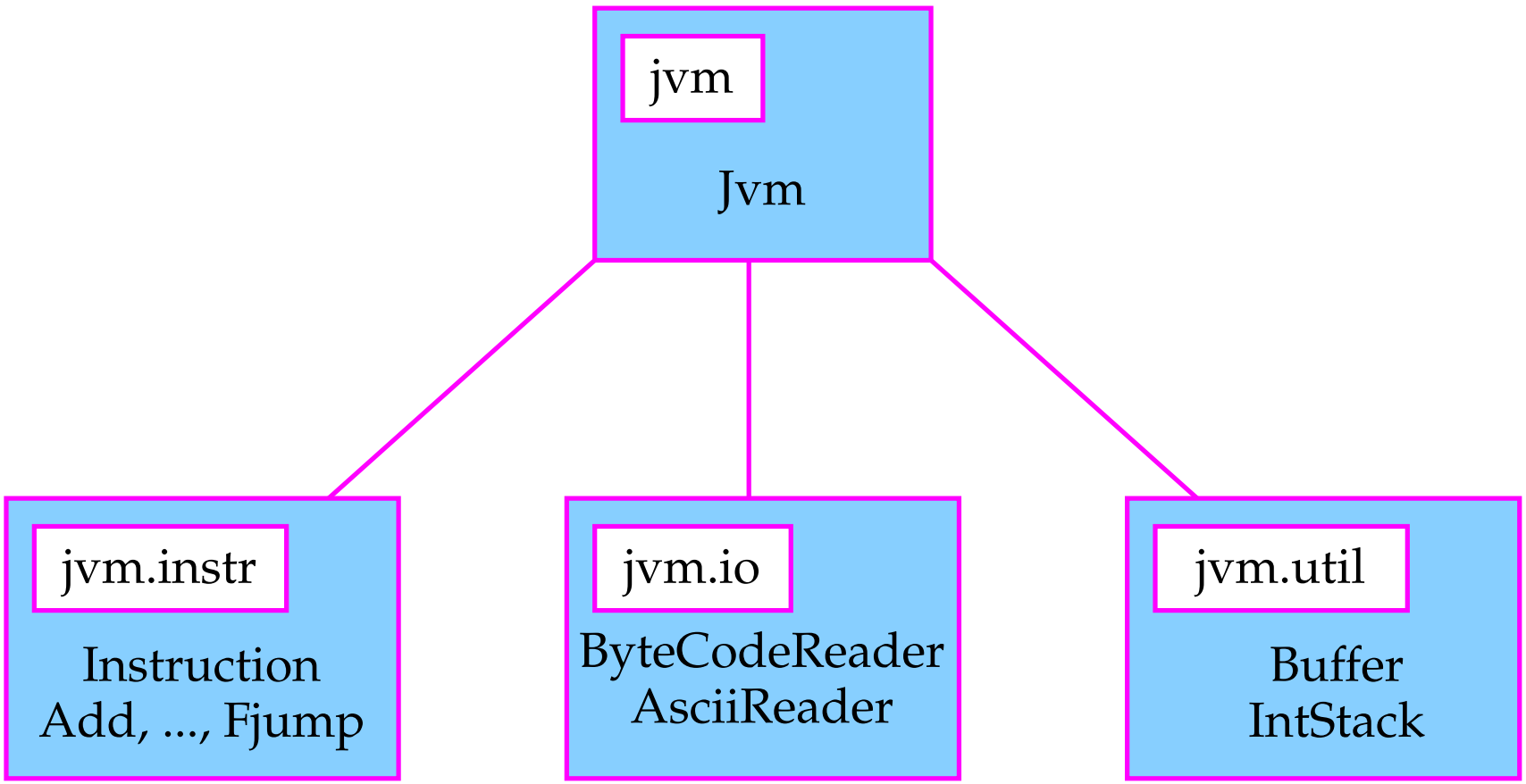


- Wünsche können sein:
  - Beseitigen von Fehlern;
  - Erweiterung der Funktionalität;
  - Portierung auf eine andere Plattform;
  - ...
- Die Leute, die die Wartung vornehmen, sind i.a. verschieden von denen, die das System implementierten.
- Gute Wartbarkeit ergibt sich aus
  - einem klaren Design;
  - einer übersichtlichen Strukturierung $\Longrightarrow$ packages;
  - einer sinnvollen, verständlichen Dokumentation.

Programm-Pakete in Java

- Ein großes System sollte hierarchisch in Teilsysteme zerlegt werden.
- Jedes Teilsystem bildet ein Paket oder `package` ...
- und liegt in einem eigenen Verzeichnis.

Beispiel: Unsere JVM


jvm
Jvm
jvm.instr
Instruction
Add, ..., Fjump
jvm.io
ByteCodeReader
AsciiReader
jvm.util
Buffer
IntStack

- Für jede Klasse muss man angeben:
  1. zu welchem Paket sie gehört;
  2. welche Pakete bzw. welche Klassen aus welchen Paketen sie verwendet.

Im Verzeichnis `a` liege die Datei `A.java` mit dem Inhalt:

```
package a;
import a.d.*;
import a.b.c.C;
class A {
   public static void main(String[] args) {
      C c = new C();
      D d = new D();
      System.out.println(c+ "   "+d);
   }
}
```

- Jede Datei mit Klassen des Pakets pckg muss am Anfang gekennzeichnet sein mit der Zeile `package pckg;`
- Die Direktive `import pckg.*;` stellt sämtliche öffentlichen Klassen des Pakets pckg den Klassen in der aktuellen Datei zur Verfügung – nicht dagegen die Unterverzeichnisse :-| .
- Die Direktive `import pckg.Cls;` stellt dagegen nur die Klasse Cls des Pakets pckg (d.h. genauer die Klasse pckg.Cls) zur Verfügung.

In den Unterverzeichnissen `b`, `b/c` und `d` von `a` liegen Dateien mit den Inhalten: