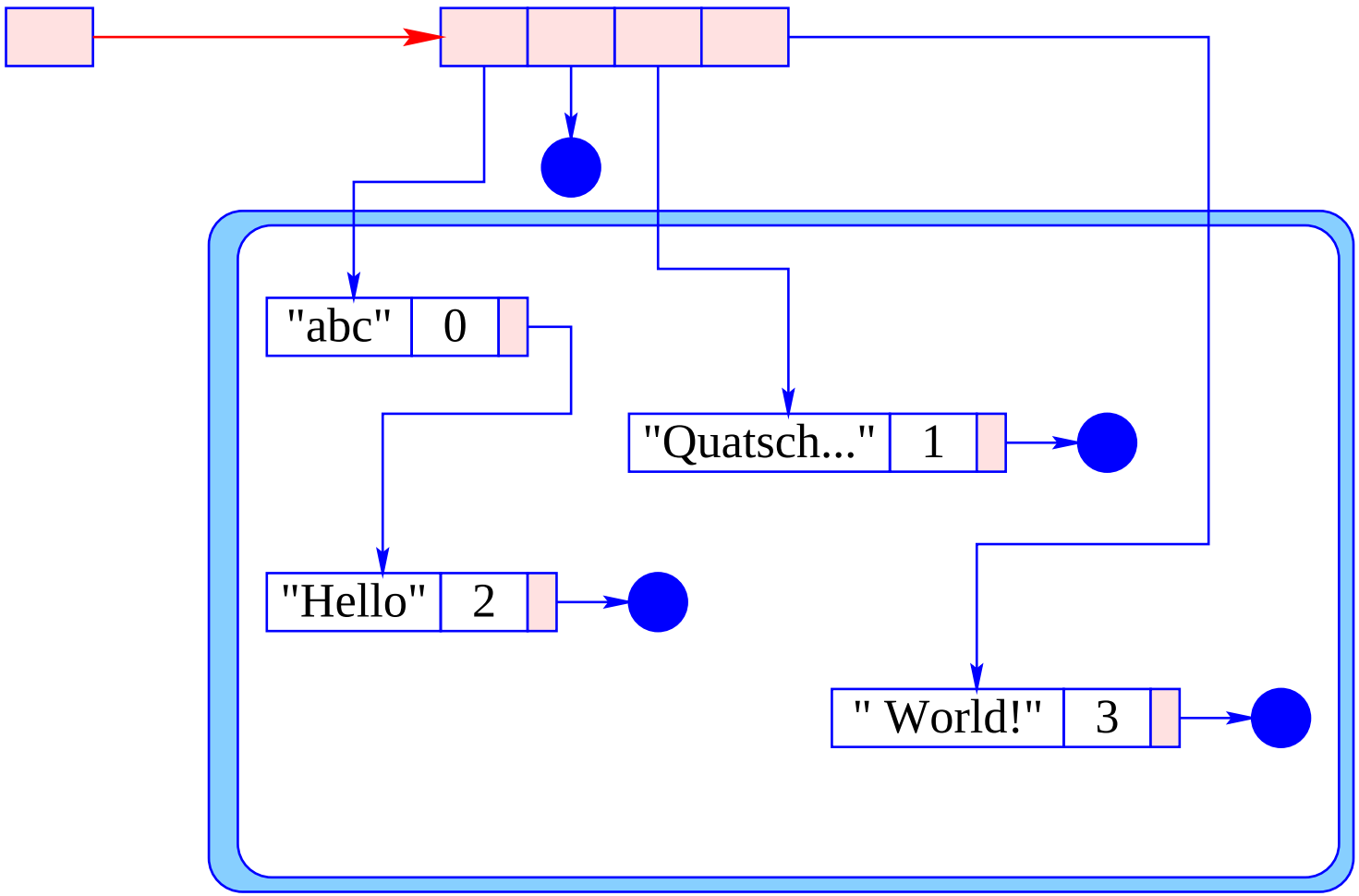
Mögliche Implementierung von `intern()`:

```
public class String {
   private static int n = 1024;
   private static List<String>[] hashSet = new List<String>[n];
   public String intern() {
      int i = (Math.abs(hashCode())%n;
      for (List<String> t=hashSet[i]; t!=null; t=t.next)
         if (equals(t.info)) return t.info;
      hashSet[i] = new List<String>(this, hashSet[i]);
      return this;
   }  // end of intern()
   ...
}     // end of class String
```

- Die Methode `hashCode()` existiert für sämtliche Objekte.
- Folglich können wir (wenn wir Lust haben :-) ähnliche Pools auch für andere Klassen implementieren.
- Vorsicht! In den Pool eingetragene Objekte können vom Garbage-Collector nicht eingesammelt werden ... :-|
- Statt nur nachzusehen, ob ein Wort `str` (bzw. ein Objekt `obj`) im Pool enthalten ist, könnten wir im Pool auch noch einen Wert hinterlegen

  $\Longrightarrow$ Implementierung von beliebigen Funktionen `String -> type` (bzw. `Object -> type`)

hashTable



Weitere Klassen zur Manipulation von Zeichen-Reihen:

- `StringBuffer` – erlaubt auch destruktive Operationen, z.B. Modifikation einzelner Zeichen, Einfügen, Löschen, Anhängen ...
- `java.util.StringTokenizer` – erlaubt die Aufteilung eines `String`-Objekts in Tokens, d.h. durch Separatoren (typischerweise White-Space) getrennte Zeichen-Teilfolgen.

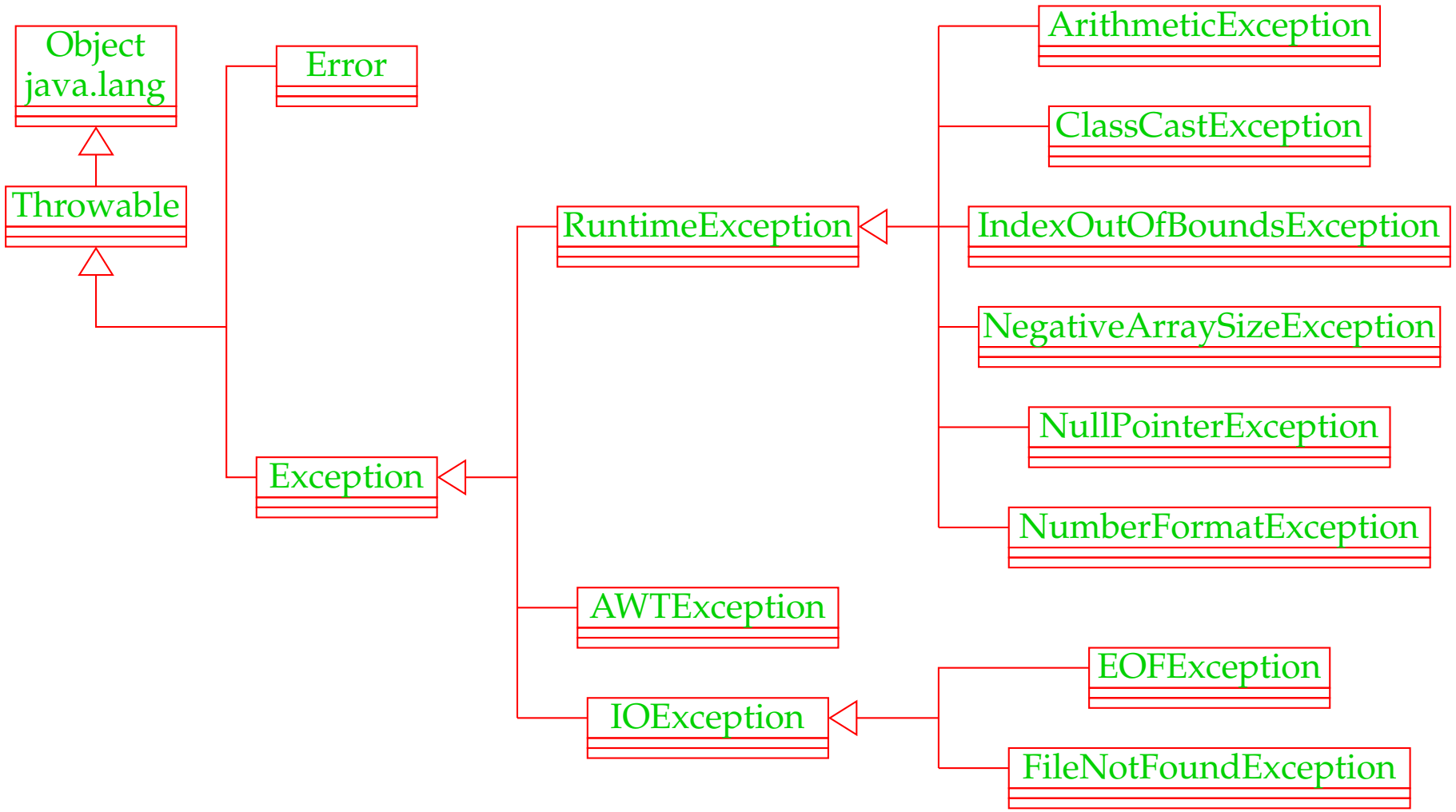
18 Fehler-Objekte: Werfen, Fangen, Behandeln

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

Idee: Explizite Trennung von

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

Einige der vordefinierten Fehler-Klassen:



Die direkten Unterklassen von `Throwable` sind:

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

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

Achtung:

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programm-Ausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten ...
- Sie braucht darum nicht im Kopf von Methoden deklariert zu werden.
- Sie kann, muss aber nicht abgefangen werden :-)

Achtung:

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programm-Ausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten ...
- Sie braucht darum nicht im Kopf von Methoden deklariert zu werden.
- Sie kann, muss aber nicht abgefangen werden :-)

Arten der Fehler-Behandlung:

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

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 ...

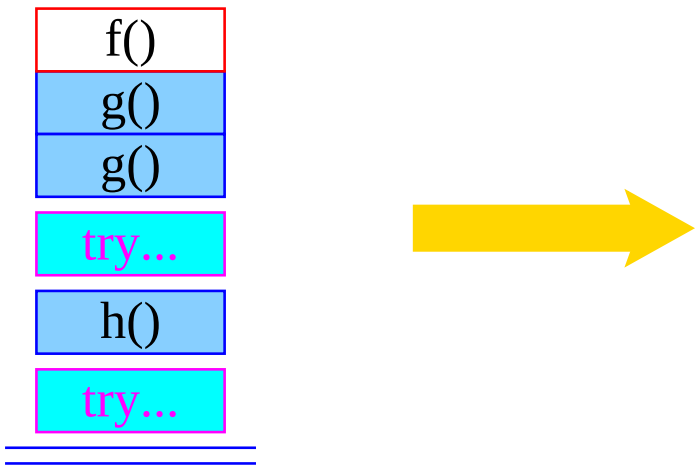
```
        while (true) {
            System.out.print(str);
            System.out.flush();
            try {
                return Integer.parseInt(stdin.readLine());
            } catch (NumberFormatException e) {
                System.out.println("Falsche Eingabe! ...");
            } catch (IOException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

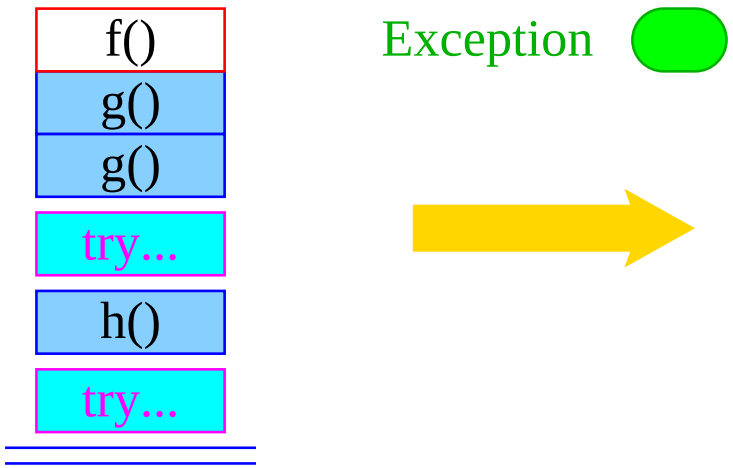
... ermöglicht folgenden Dialog:

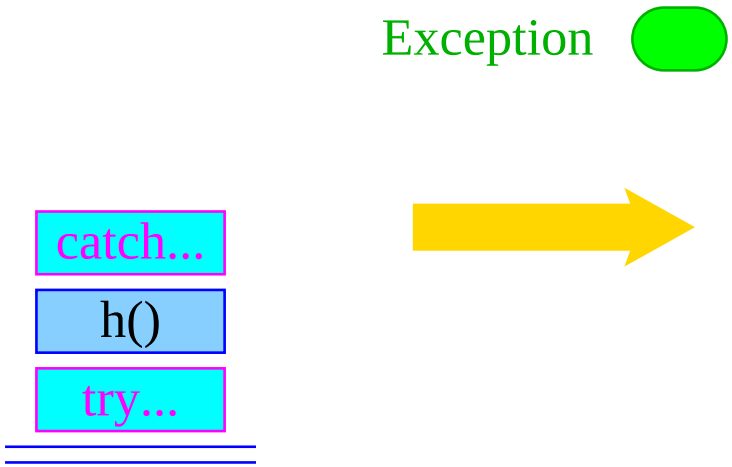
```
> java Adding
1. Zahl:          abc
Falsche Eingabe! ...
1. Zahl:          0.3
Falsche Eingabe! ...
1. Zahl:          17
2. Zahl:          25
Summe:            42
```

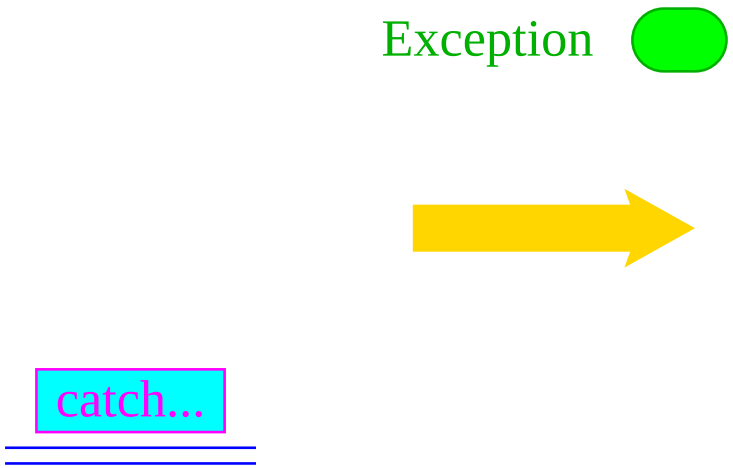
- Ein Exception-Handler besteht aus einem `try`-Block `try {` ss `}` , in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.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