- 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 Programmierfehler und ihre Behebung

(kleiner lebenspraktischer Ratgeber)

Grundsätze:

- Jeder Mensch macht Fehler :-)
- ... insbesondere beim Programmieren.
- Läuft ein Programm, sitzt der Fehler tiefer.
- Programmierfehler sind Denkfehler.
- Um eigene Programmierfehler zu entdecken, muss nicht ein Knoten im Programm, sondern ein Knoten im Hirn gelöst werden.

18.1 Häufige Fehler und ihre Ursachen

- Das Programm terminiert nicht.

  Mögliche Gründe:

  - In einer Schleife wird die Schleifen-Variable nicht modifiziert.

```
...
String t = file.readLine();
while(t! = null)
   System.out.println(t);
...
```

- In einer Rekursion fehlt die Abbruch-Bedingung.

```
public static int find0(int[] a, int x, int l, inr r) {
   int t = (l+r)/2;
   if (x<=a[t]) return find0(a,x,l,t);
   return find0(a,x,t+1,r);
   }
```

- Das Programm wirft eine `NullPointerException`.

  Möglicher Grund:

  Eine Objekt-Variable wird benutzt, ohne initialisiert zu sein:

  - ... weil sie in einem Feld liegt:

```
Stack[] h = new Stack[4];
...
for(int i=0; i<4; ++i)
   h[i].push(i);
....
```

- ... oder einem Objekt ohne passenden Konstruktor:

```
import java.io.*;
class A {
   public A a;
   }
class AA {
   public static void main(String[] args) {
      A aa = (new A()).a;
      System.out.println(aa);
      System.out.println(aa.a);
   }
}
```

- Eine Instanz-Variable verändert auf geheimnisvolle Weise ihren Wert.

  Möglicher Grund:

  Es gibt weitere Verweise auf das Objekt mit (unerwünschten?) Seiteneffekten ...

```
...
List l1 = new List(3);
List l2 = l1;
l2.info = 7;
...
```

- Ein Funktionsaufruf hat überhaupt keinen Effekt ...

```
public static void reverse (String [] a) {
   int n = a.length();
   String [] b = new String [n];
   for (int i=0; i<n; ++i) b[i] = a[n-i-1];
   a = b;
}
```

- Ein Funktionsaufruf hat überhaupt keinen Effekt ...

```
public static void reverse (String [] a) {
   int n = a.length();
   String [] b = new String [n];
   for (int i=0; i<n; ++i) b[i] = a[n-i-1];
   a = b;
}
```

- Eine bedingte Verzweigung liefert merkwürdige Ergebnisse.

  Mögliche Gründe:

  - `equals()` mit == verwechselt?
  - Die `else`-Teile falsch organisiert?

18.2 Generelles Vorgehen zum Testen von Software

(1) Feststellen fehlerhaften Verhaltens.

Problem: Auswahl einer geeigneter Test-Scenarios

Black-Box Testing: Klassifiziere Benutzungen!

Finde Repräsentanten für jede (wichtige) Klasse!

White-Box Testing: Klassifiziere Berechnungen – z.B. nach

- besuchten Programm-Punkten,
- benutzten Datenstrukturen oder Klassen
- benutzten Methoden, geworfenen Fehler-Objekten ...

Finde repräsentantive Eingabe für jede (wichtige) Klasse!

Beispiel: `int find(int[] a, int x);`

Black-Box Test: Klassifizierung denkbarer Argumente:

1. `a == null;`
2. `a != null` :

2.1. `x` kommt in `a` vor $\Longrightarrow$ `a == [42], x == 42`

2.2. `x` kommt nicht in `a` vor $\Longrightarrow$ `a == [42], x == 7`

Achtung:

Nicht in allen Klassen liefert `find()` sinnvolle Ergebnisse ...

$\Longrightarrow$ Überprüfe, ob alle Benutzungen in sinnvolle Klassen fallen :-)

White-Box Test: Klassifizierung von Berechnungen: