

Vorlesungsteil

Debuggen und Fehlerbehandlungen



Wer einen Fehler gemacht hat und ihn nicht korrigiert, begeht einen zweiten.
Konfuzius

Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

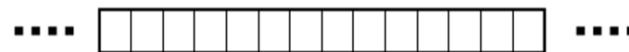
Debuggen und Fehlerbehandlungen

- Typische Fehlerklassen
- Vor- und Nachbedingungen
- Zusicherungen (assertions)
- Exceptions fangen
- Fehlerort lokalisieren
- Exceptions an Aufrufer weiterleiten
- Projekt: PI
- Textausgaben in der Konsole
- Zeilenweises Debuggen und Breakpoints
- Entwicklungsumgebung Eclipse
- Testgetriebene Softwareentwicklung
- Arbeiten mit JUnit
- Annotationen und Vergleichsmethoden
- Refactoring
- Checklist für TDD

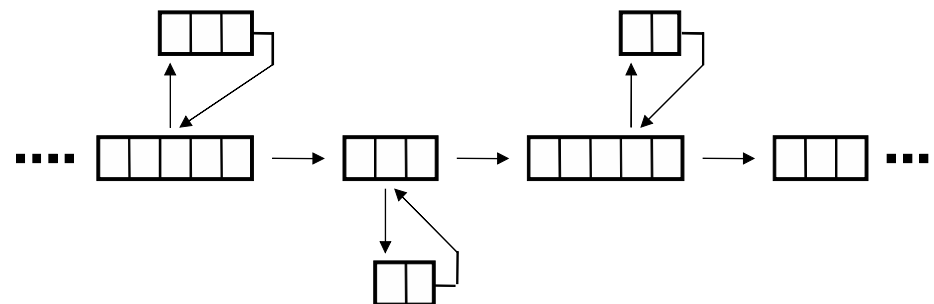


Das richtige Konzept

Es sollte bei der Entwicklung darauf geachtet werden, dass nicht allzu viele Programmzeilen zu einer Funktion gehören. Das Problem sind oft wieder-verwendete Variablen oder Seiteneffekte:



Besser ist es, das Programm zu gliedern und thematisch in Programmabschnitte zu unterteilen. Zum Einen wird damit die Übersicht gefördert und zum Anderen verspricht die Modularisierung den Vorteil, Fehler in kleineren Programmabschnitten besser aufzuspüren und vermeiden zu können:



Beim Conway-Projekt haben wir es bereits richtig vorgemacht.

Typisch auftretende Fehlerklassen

Wir unterscheiden hauptsächlich zwischen diesen drei Fehlerklassen:

1. **Kompilierfehler:** Die Syntaxregeln der Sprache wurden nicht eingehalten, der Java-compiler beschwert sich.
2. **Laufzeitfehler:** Im laufenden Programm werden Regeln verletzt, die zum Absturz führen können.
3. **Inhaltliche Fehler:** Das Programm tut nicht das Richtige, es gibt logische Fehler.

Am Anfang scheint es manchmal ausweglos zu sein, **Kompilierfehler** ohne Hilfe zu beheben. Aber mit einigen Beispielen und genügend Erfahrung sind das eigentlich die Fehler, die am schnellsten zu lösen sind, denn Java unterstützt uns mit den Fehlermeldungen enorm.

Laufzeitfehler sind schon unangenehmer, da sie oft nur unter bestimmten Voraussetzungen entstehen und teilweise nicht reproduzierbar sind. Solche Fehler müssen wir sehr Ernst nehmen und bereits beim Entwurf die typischen Fallstricke erkennen.

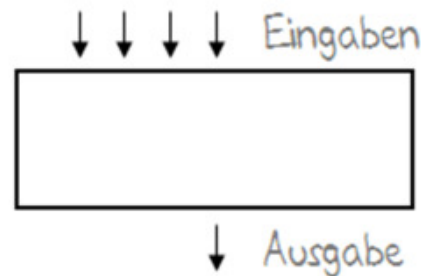
Logische Fehler sind am schwierigsten zu identifizieren. Was ist also, wenn zulässige Eingaben nicht zu gewünschten Ergebnissen führen? Nach einem Abschnitt über Fehlerbehandlungen in Java, in dem wir uns den Laufzeitfehlern widmen wollen, werden wir exemplarisch einen logischen Fehler Schritt für Schritt aufspüren.

Vor- und Nachbedingungen

Ein wichtiges Werkzeug der Modularisierung stellen die sogenannten **Constraints** (Einschränkungen) dar.

Beim Eintritt in einen Programmabschnitt (z.B. eine Funktion) müssen die Eingabeparameter überprüft werden, um klarzustellen, dass der Programmteil mit den Eingabedaten arbeiten kann.

Wir nennen das auch die notwendigen **Vorbedingungen** (preconditions). Wenn die Vorbedingungen erfüllt sind, muss der Programmabschnitt das Gewünschte liefern, also zugesicherte **Nachbedingungen** (postconditions) erfüllen:



Der Entwickler einer Funktion ist dafür verantwortlich, dass auch die richtigen Berechnungen durchgeführt und zurückgeliefert werden. Dies stellt sicher, dass ein innerhalb eines Moduls auftretender Fehler, auch dort gesucht und behandelt werden muss und nicht im aufrufenden Modul, welches eventuell fehlerhafte Daten geliefert hat.

Beispiel zur Berechnung der Fakultät

Als Beispiel schauen wir uns mal die Fakultätsfunktion an:

```
public class Fakultaet {
    /*
     * Fakultätsfunktion liefert für n=0 ... 20 die entsprechenden
     * Funktionswerte  $n! = n * (n-1) * (n-2) * \dots * 1$ 
     * ( $0!$  ist per Definition 1)
     *
     * Der Rückgabewert liegt im Bereich 1 .. 2432902008176640000
     *
     * Sollte eine falsche Eingabe vorliegen, so liefert das Programm
     * als Ergebnis -1.
     */
    public static long fakultaet(int n) {
        // Ist der Wert außerhalb des erlaubten Bereichs?
        if ((n < 0) || (n > 20))
            return -1;

        long erg = 1;
        for (int i = n; i > 1; i--)
            erg *= i;
        return erg;
    }

    public static void main(String[] args) {
        for (int i = -2; i < 22; i++)
            System.out.println("Fakultaet von "+i+" ist "+fakultaet(i));
    }
}
```

Live-Coding-Session

A blurred background image showing snippets of code from a document. The visible code includes:

```
number: 1;  
contents: 1;  
$count; $1++;  
put type="|", $data[0]  
$i + 1, "|";  
$i, $totalsecurity);  
cho "checked";  
if ($i == 0) {  
    ("checked");
```

Ausgabe der Fakultätsfunktion

Damit haben wir ersteinmal die Ein- und Ausgaben überprüft und sichergestellt, dass wir mit den gewünschten Daten arbeiten, nun müssen wir dafür Sorge tragen, dass die Module fehlerfrei arbeiten.

Wenn wir die Funktion `fakultaet`, wie in der `main`-Funktion gezeigt, mit verschiedenen Parametern testen, erhalten wir die folgende Ausgabe:

```
C:\Java>java Fakultaet
Fakultaet von -2 ist -1
Fakultaet von -1 ist -1
Fakultaet von 0 ist 1
Fakultaet von 1 ist 1
Fakultaet von 2 ist 2
Fakultaet von 3 ist 6
Fakultaet von 4 ist 24
...
Fakultaet von 18 ist 6402373705728000
Fakultaet von 19 ist 121645100408832000
Fakultaet von 20 ist 2432902008176640000
Fakultaet von 21 ist -1
```

Wir sehen, dass das Programm für unzulässige Eingaben wie erwartet eine -1 zurückliefert. Bei diesem Beispiel ist das machbar, denn die Fakultätsfunktion liefert nach ihrer Definition nur positive Werte.

Aber was machen wir, wenn wir beispielsweise eine Funktion erstellen wollen, die sowohl positive als auch negative Werte zurückliefern kann? Zu dieser Frage kommen wir zu einem späteren Zeitpunkt noch einmal zurück.

Zusicherungen in Java

Gerade haben wir im Kommentarteil zu einer Funktion die Bedingungen für die Ein- und Ausgaben schriftlich festgehalten. Anstatt nun mit vielen if-Anweisungen, die Inhalte zu prüfen, können wir ein in Java vorhandenes Konzept verwenden.

So können wir Zusicherungen (assertions) zur Laufzeit eines Programms mit der folgenden Anweisung erzwingen:

```
assert <Bedingung>;
```

Sollte die Bedingung nicht erfüllt sein, bricht das Programm mit einer Fehlermeldung ab. Damit lassen sich auch logische Fehler in der Entwicklungsphase besser vermeiden und identifizieren.

Damit Java die Überprüfung der Zusicherungen durchführt, müssen wir das Programm mit einem kleinen Zusatz starten:

```
java -ea <Programmname>
```

Fakultätsfunktion und Zusicherungen

Um unsere Bedingungen für die Eingabe (n aus dem Intervall $[0, 20]$) und die Ausgabe (Ergebnis ist positiv), könnten wir beispielsweise wie folgt als Zusicherungen in die Funktion einbauen:

```
public static long fakultaet(int n) {  
    // Ist der Wert außerhalb des erlaubten Bereichs?  
    assert ((n >= 0) && (n <= 20)); // Vorbedingung (precondition)  
  
    long erg = 1;  
    for (int i = n; i > 1; i--)  
        erg *= i;  
  
    assert erg>0; // Nachbedingung (postcondition)  
    return erg;  
}
```

Wenn wir das veränderte Programm jetzt mit der Option `-ea` (enable assertions) starten, dann erhalten wir:

```
C:\Java>java -ea Fakultaet  
Exception in thread "main" java.lang.AssertionError  
    at FakultaetAssertion.fakultaet(FakultaetAssertion.java:15)  
    at FakultaetAssertion.main(FakultaetAssertion.java:27)
```

Wir erkennen zwar, dass die Zusicherung in Zeile 15 zum Abbruch geführt hat, es wäre aber sicherlich hilfreicher, wenn die Fehlermeldung einen aussagekräftigen Hinweis enthält.

Fakultätsfunktion und "sprechende" Zusicherungen

Wir können folgende Syntax verwenden, die um eine Fehlermeldung erweitert ist:

`assert <Bedingung>: <Fehlermeldung>;`

Wir könnten die Vorbedingung beispielsweise so abändern:

```
assert ((n >= 0) && (n <= 20)): "Eingabe n nicht zulaessig";
```

Die Ausgabe der Fehlermeldung ist jetzt sehr viel lesbarer.

Wir haben an dieser Stelle das Konzept der Zusicherungen sowohl für die Vor- als auch die Nachbedingungen verwendet. Es gehört aber zum guten Programmierstil, nur die Nachbedingungen mit Hilfe von `assert` zu prüfen.

Jetzt werden wir Exceptions kennenlernen, die sich für die Einhaltung der Vorbedingungen besser eignen.

Konzept der Exceptions

Wenn ein Fehler während der Ausführung eines Programms auftritt, wird ein Objekt einer Fehlerklasse (Exception) erzeugt.

Da der Begriff Objekt erst später erläutert wird, stellen wir uns einfach vor, dass ein Programm gestartet wird, welches den Fehler analysiert und wenn der Fehler identifizierbar ist, können wir dieses Programm nach dem Fehler fragen und erhalten einen Hinweis, der Aufschluss über die Fehlerquelle Fehlerquelle gibt

Schauen wir uns ein Beispiel mit einer Problematik an, die uns schon einmal begegnet ist:

```
public class ExceptionTest{
    public static void main(String[] args){
        int d = Integer.parseInt(args[0]);
        int k = 10/d;
        System.out.println("Ergebnis ist "+k);
    }
}
```

Der Aufruf `Integer.parseInt` wandelt eine Zeichenkette in die entsprechende Zahl um.

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms.

Fehler über Fehler

Wir erhalten teilweise Laufzeitfehler bei den folgenden Eingaben:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionTest.main(ExceptionTest:5)

C:\>java ExceptionTest d
Exception in thread "main" java.lang.NumberFormatException: For input
string: "d"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at ExceptionTest.main(ExceptionTest:4)
```

Zwei Fehlerquellen sind hier erkennbar, die Eingabe eines falschen Datentyps und die Eingabe einer 0, die bei der Division durch 0 einen Fehler verursacht. Beides sind für Java wohlbekannte Fehler, daher gibt es auch in beiden Fällen entsprechende Fehlerbezeichnungen `NumberFormatException` und `ArithmeticException`.

Wir Fragen uns: In welchen Situationen bzw. unter welchen Bedingungen stürzt das Programm ab?

Einfache try-catch-Behandlung

Die Syntax dieser Klausel sieht wie folgt aus:

```
try {  
    <Anweisung>;  
    ...  
    <Anweisung>;  
} catch(Exception e){  
    <Anweisung>;  
}
```

Die try-catch-Behandlung lässt sich lesen als: Versuche dies, wenn ein Fehler dabei auftritt, mache jenes.

Einsatz von try-catch (einfach)

Um unser Programm vor einem Absturz zu bewahren, wenden wir diese Klausel an:

```
try {  
    int d = Integer.parseInt(args[0]);  
    int k = 10/d;  
    System.out.println("Ergebnis ist "+k);  
} catch (Exception e) {  
    System.out.println("Fehler ist aufgetreten...");  
}
```

Wie testen nun die gleichen Eingaben, die vorhin zu Fehlern geführt haben:

```
C:\>java ExceptionTest 2  
Ergebnis ist 5  
  
C:\>java ExceptionTest 0  
Fehler ist aufgetreten...  
  
C:\>java ExceptionTest d  
Fehler ist aufgetreten...
```

Einen Teilerfolg haben wir schon zu verbuchen, da das Programm mit den fehlerhaften Eingaben zur Laufzeit nicht mehr abstürzt.

Mehrfache try-catch-Behandlung

Dummerweise können wir die auftretenden Fehler nicht mehr eindeutig identifizieren, da sie jeweils die gleichen Fehlermeldungen produzieren.

Um die Fehler aber eindeutig zu unterscheiden und behandeln zu können, lassen sich einfach mehrere catch-Blöcke mit verschiedenen Fehlertypen angeben:

```
try {  
    <Anweisung>;  
    ...  
    <Anweisung>;  
} catch(Exceptiontyp1 e1){  
    <Anweisung>;  
} catch(Exceptiontyp2 e2){  
    <Anweisung>;  
} catch(Exceptiontyp3 e3){  
    <Anweisung>;  
} finally {  
    <Anweisung>;  
}
```

Der Vollständigkeit halber wollen wir noch erwähnen, dass am Ende ein optionaler finally-Block folgen kann, der Anweisungen enthält, die in jedem Fall am Ende ausgeführt werden.

Einsatz von try-catch (mehrfach)

Wenden wir die neue Erkenntnis auf unser Programm an und erweitern es:

```
try {  
    int d = Integer.parseInt(args[0]);  
    int k = 10/d;  
    System.out.println("Ergebnis ist "+k);  
} catch (NumberFormatException nfe){  
    System.out.println("Falscher Typ! Gib eine Zahl ein ...");  
} catch (ArithmeticException ae){  
    System.out.println("Division durch 0! ...");  
} catch (Exception e){  
    System.out.println("Unbekannter Fehler aufgetreten ...");  
}
```

Wie testen nun die gleichen Eingaben, die vorhin zu Fehlern geführt haben:

```
C:\>java ExceptionTest3 2  
Ergebnis ist 5  
  
C:\>java ExceptionTest3 0  
Division durch 0! ...  
  
C:\>java ExceptionTest3 d  
Falscher Typ! Gib eine Zahl ein ...
```

Den Fehlerort lokalisieren

Wenn wir in einem Programm zur Laufzeit einen Fehler erzeugen, dann liefert uns Java den genauen Ort. Hier haben wir beispielsweise eine Methode `erzeugeFehler`, die ganz offensichtlich zu einem Fehler führt:

```
public class ExceptionLokalisieren {  
    public static void erzeugeFehler() {  
        int erg = 3/0;  
    }  
  
    public static void rufeFehlerAuf() {  
        erzeugeFehler();  
    }  
  
    public static void main(String[] args) {  
        rufeFehlerAuf();  
    }  
}
```

Das führt zur Laufzeit den folgenden Fehler:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at ExceptionLokalisieren.erzeugeFehler(ExceptionLokalisieren.java:5)  
at ExceptionLokalisieren.rufeFehlerAuf(ExceptionLokalisieren.java:9)  
at ExceptionLokalisieren.main(ExceptionLokalisieren.java:13)
```

Den Fehlerort lokalisieren

Sollten wir den Aufruf nun in der main-Methode durch einen try-catch-Block einschließen, geht diese Fehlermeldung zunächst verloren, kann aber durch die Methode `printStackTrace` wieder ausgegeben werden:

```
public class ExceptionLokalisieren {  
    public static void erzeugeFehler() {  
        int erg = 3/0;  
    }  
  
    public static void rufeFehlerAuf() {  
        erzeugeFehler();  
    }  
  
    public static void main(String[] args) {  
        try {  
            rufeFehlerAuf();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Fehler an Aufrufer weiterreichen

Eine Funktion liefert immer genau ein Ergebnis. Wir können allerdings zusätzlich noch erlauben, dass konkrete Fehler zurückgeliefert werden können. Wenn wir eine Funktion derart ausstatten wollen, erweitern wir die Funktionssignatur (im Beispiel aufgrund des Platzmangels ohne Parameterliste) wie folgt:

```
public static <Datentyp> name() throws Exception {  
    // ...  
    throw new Exception(<Fehlermeldung>);  
}
```

Wenn uns jetzt beispielweise ein Eingabefehler auffällt, können wir die Funktion jederzeit durch throw mit einer Fehlermeldung beenden.

Fakultätsfunktion mit Exception

```
public class FakultaeException {
    /*
     * Fakultätsfunktion liefert für n=0 ... 20 die entsprechenden
     * Funktionswerte  $n! = n * (n-1) * (n-2) * \dots * 1$ 
     * (0! ist per Definition 1)
     *
     * Der Rückgabewert liegt im Bereich 1 .. 2432902008176640000
     *
     * Sollte eine falsche Eingabe vorliegen, so wirft das Programm
     * an den Aufrufer einen Fehler zurück.
     */
    public static long fakultaet(int n) throws Exception {
        // Ist der Wert außerhalb des erlaubten Bereichs?
        if ((n < 0) || (n > 20))
            throw new Exception("AUSSERHALB DES ZULAESSIGEN BEREICHS");

        long erg = 1;
        for (int i = n; i > 1; i--)
            erg *= i;
        return erg;
    }

    public static void main(String[] args) {
        for (int i = -2; i < 22; i++) {
            try {
                System.out.println("Fakultaet von "+i+" ist "+fakultaet(i));
            } catch (Exception e){
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Testen der Fakultätsfunktion

Wir erhalten die hier etwas gekürzte Konsolenausgabe:

```
AUSSERHALB DES ZULAESSIGEN BEREICHS  
AUSSERHALB DES ZULAESSIGEN BEREICHS  
Fakultaet von 0 ist 1  
Fakultaet von 1 ist 1  
...  
Fakultaet von 20 ist 2432902008176640000  
AUSSERHALB DES ZULAESSIGEN BEREICHS
```

Live-Coding-Session

A blurred background image showing snippets of code from a document. The visible code includes:

```
number: 1;  
contents: 1;  
$count; $1++;  
put type="|", $data[0]  
$i + 1, "|";  
$i, $totalsecurity);  
cho "checked";  
if ($i == 0) {  
    ("checked");
```

Fehlerhafte Berechnungen aufspüren

Beim Programmieren wird leider ein wesentlicher Teil der Zeit mit dem Aufsuchen von Fehlern verbracht.

Das ist selbst bei sehr erfahrenen Programmierern so und gerade, wenn die Projekte größer und unübersichtlicher werden, sind effiziente Programmiertechniken, die Fehler vermeiden, unabdingbar.

Sollte sich aber doch ein Fehler eingeschlichen haben, gibt es einige Vorgehensweisen, die die zum Auffinden benötigte Zeit auf ein Mindestmaß reduzieren.

Textausgaben auf der Konsole

An der einen und anderen Stelle, konnten wir bereits erfolgreich Informationen auf der Konsole ausgeben. Jetzt wollen wir das etwas genauer spezifizieren. Es gibt prinzipiell drei Möglichkeiten, Daten auszugeben.

Der Unterschied zwischen der ersten und zweiten Variante ist lediglich, dass die Ausgabe mit einem Zeilenumbruch abgeschlossen wird:

```
System.out.print(<String>);  
System.out.println(<String>);
```

Für die ersten beiden Varianten gilt: Ein <String> kann dabei aus verschiedenen Elementen von Zeichenketten und Ausdrücken bestehen, die durch ein + verknüpft (konkateniert) werden:

```
boolean b = true;  
int i = 10, j = -5;  
double d = 41.6229;  
  
System.out.println(d);  
System.out.println("Text " + b);  
System.out.println("Text " + i + " Text" + j);
```

Wenn wir als Ausdruck nicht nur eine Variable zu stehen haben, müssen wir darauf achten, dass der Ausdruck geklammert ist:

```
System.out.println("Text " + (i + 1));
```

Besondere Variante aus C/C++

Bei der dritten Variante handelt es sich um eine formatierte Ausgabe:

```
System.out.printf(<String>, <Datentyp1>, <Datentyp2>, ...);
```

Wir konstruieren eine Zeichenkette <String> und vergeben dort Platzhalter, beginnend mit dem Symbol %. Anschließend werden diese in der Reihenfolge eingesetzt, wie sie nach der Zeichenkette angegeben werden. Sollten diese nicht übereinstimmen, wird eine Fehlermeldung geliefert.

Die Platzhalter spezifizieren gleich den Datentyp, so stehen beispielsweise

- b für Boolean,
- d für Integer,
- o für Oktalzahl,
- x für Hexadezimalzahl,
- f für Gleitkommazahl,
- e für Exponentenschreibweise,
- s für Zeichenkette und
- n für Zeilenumbruch.

Beispiele

Schauen wir uns gleich Beispiele an:

```
System.out.printf("Text %d, %d, %b \n", i, j, b);  
System.out.printf("Text >%10.3f< \n", d);  
System.out.printf("Text >0123456789<", d);
```

Dass es sich um eine formatierte Ausgabe handelt, sehen wir im zweiten Beispiel. Hier reservieren wir für die Ausgabe der Zahl 10 Plätze und geben eine Genauigkeit von 3 Stellen nach dem Komma an.

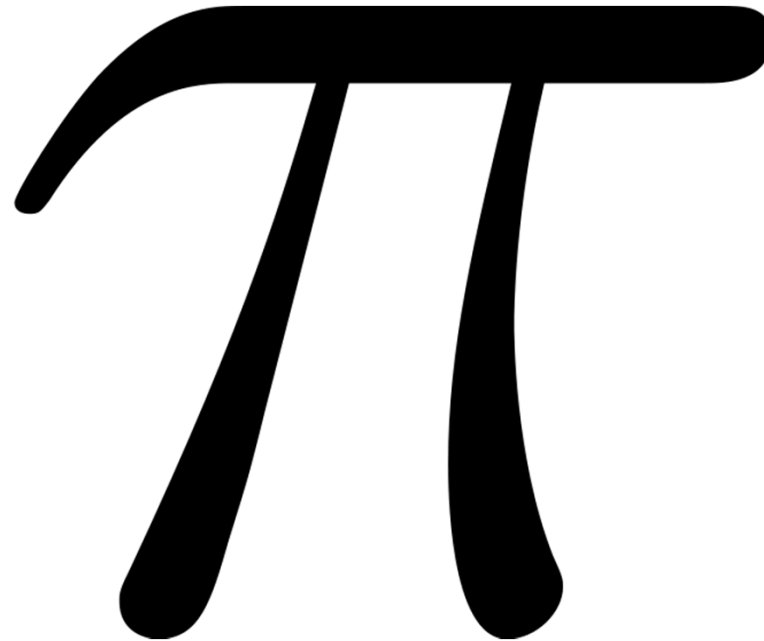
Als Ausgabe erhalten wir:

```
Text 10, -5, true  
Text >    41,623<  
Text >0123456789<
```

Wir sehen in der zweiten Zeile, wie die 10 Plätze rechtsbündig aufgefüllt wurden.

In den Zeichenketten können auch Steuerzeichen (Escape-Sequenzen) vorkommen. Wir kennen bereits `\n` für einen Zeilenumbruch und `\t` für einen Tabulator.

Das nächste Projekt!



~~political incorrent~~

Kreiszahl

Näherung der Kreiszahl pi nach Madhava-Leibniz

Gottfried Wilhelm Leibniz gab 1682 eine Berechnungsvorschrift für die Näherung von $\pi/4$ an. Da aber der indische Gelehrte Mādhava of Saṅgamāgrama bereits im 14. Jahrhundert diese Reihe entdeckte, ist diese als Madhava-Leibniz-Reihe bekannt.

Die Vorschrift der Reihe besagt:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots = \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

Der Nenner wird also immer um 2 erhöht, während das Vorzeichen jeden Schritt wechselt. Eine Schleife bietet sich zur Berechnung an. Am Ende müssen wir das Ergebnis schließlich noch mit 4 multiplizieren, um eine Näherung für pi zu erhalten.

Programm mit kleinem Fehler

Unser erster Versuch:

```
public class MadhavaLeibnizReihe {  
    public static double piMadhavaLeibniz(int maxIterationen){  
        double piViertel = 0;  
        int vorzeichen = 1;  
        for (int i=1; i<=maxIterationen*2; i+=2){  
            piViertel += vorzeichen*(1/i);  
            vorzeichen *= -1;  
        }  
        return 4*piViertel;  
    }  
  
    public static void main(String[] args){  
        double piLeibniz = piMadhavaLeibniz(100);  
        System.out.println("Eine Naehierung fuer PI ist "+piLeibniz);  
    }  
}
```

Die Kreiszahl pi beginnt mit dem Präfix (Anfangsteil) 3.141592..., wir erwarten nach 100 Iterationen also ein ähnliches Ergebnis. Stattdessen erhalten wir:

```
C:\JavaCode>java MadhavaLeibnizReihe  
Eine Naehierung fuer PI ist 4.0
```

Mit dem Wert 4.0 haben wir keine wirklich gute Näherung.

Wer hat den Fehler bereits entdeckt?

Anekdote zu pi und zweiter Test

Eine oft angegebene Anekdote in diesem Zusammenhang besagt, dass 1897 im US-Bundesstaat Indiana dieser Wert für pi sogar per Gesetz festgelegt wurde. Beim Studieren dieses Gesetzes wird allerdings klar, dass der Wert gar nicht auf 4 sondern auf 3.2 festgelegt wurde.

Erhöhen wir die Iterationsanzahl von 100 auf 1000, bleibt das Ergebnis ebenfalls unverändert.

Zeilenweises Debuggen I

Um den Fehler aufzuspüren, versuchen wir die Berechnungen schrittweise nachzuvollziehen. Überprüfen wir zunächst, ob das Vorzeichen und der Nenner für die Berechnung stimmen. Dazu fügen wir in die Schleife folgende Ausgabe ein:

```
for (int i=1; i<maxIterationen; i+=2){  
    System.out.println("i:"+i+" vorzeichen:"+vorzeichen);  
    piViertel += vorzeichen*(1/i);  
    vorzeichen *= -1;  
}
```

Zwischenausgaben helfen uns, die Abarbeitung besser nachzuvollziehen. Als Ausgabe erhalten wir:

```
C:\JavaCode>java MadhavaLeibnizReihe  
i:1 vorzeichen:1  
i:3 vorzeichen:-1  
i:5 vorzeichen:1  
i:7 vorzeichen:-1  
...
```

Das Vorzeichen alterniert korrekt, ändert sich also bei jeder Iteration.

Zeilenweises Debuggen II

Die Ausgabe erweitern wir jetzt, um zu sehen, wie sich `piViertel` im Laufe der Berechnungen verändert:

```
System.out.println("i:"+i+" vorzeichen:"+vorzeichen+"\t piViertel:"+piViertel);
```

Wir erhalten die folgende interessante Ausgabe:

```
C:\JavaCode>java MadhavaLeibnizReihe
i:1 vorzeichen:1      piViertel:0.0
i:3 vorzeichen:-1     piViertel:1.0
i:5 vorzeichen:1      piViertel:1.0
i:7 vorzeichen:-1     piViertel:1.0
...
```

Vor dem ersten Schritt hat `piViertel` den Initialwert 0. Nach dem ersten Schritt ist `piViertel=1`, soweit so gut. Allerdings ändert sich `piViertel` in den weiteren Iterationen nicht mehr.

Hier tritt der Fehler also zum ersten Mal auf.

Zeilenweises Debuggen III

Werfen wir einen Blick auf die Zwischenergebnisse:

```
double zwischenergebnis = vorzeichen*(1/i);  
System.out.println("i:"+i+" Zwischenergebnis:"+zwischenergebnis);
```

Jetzt sehen wir die Stelle, an der etwas schief gelaufen ist:

```
C:\JavaCode>java MadhavaLeibnizReihe  
i:1 Zwischenergebnis:1.0  
i:3 Zwischenergebnis:0.0  
i:5 Zwischenergebnis:0.0  
i:7 Zwischenergebnis:0.0  
...
```

Wer erkennt den Fehler jetzt?

Lösung des Problems

Die Berechnung $(1/i)$ liefert immer das Ergebnis 0, da sowohl 1 als auch i vom Typ `int` sind.

Der Compiler verwendet die `ganzzahligDivision`, was bedeutet, dass alle Stellen nach dem Komma abgerundet werden. Dieses Problem lässt sich leicht lösen, indem wir die 1 in `1.d` ändern und damit aus dem implizit angenommenen `int` ein `double` machen.

Eine andere Möglichkeit wäre das Casten von i zu einem `double`.

Wir entfernen die Ausgaben und starten das korrigierte Programm mit 1000 Iterationen:

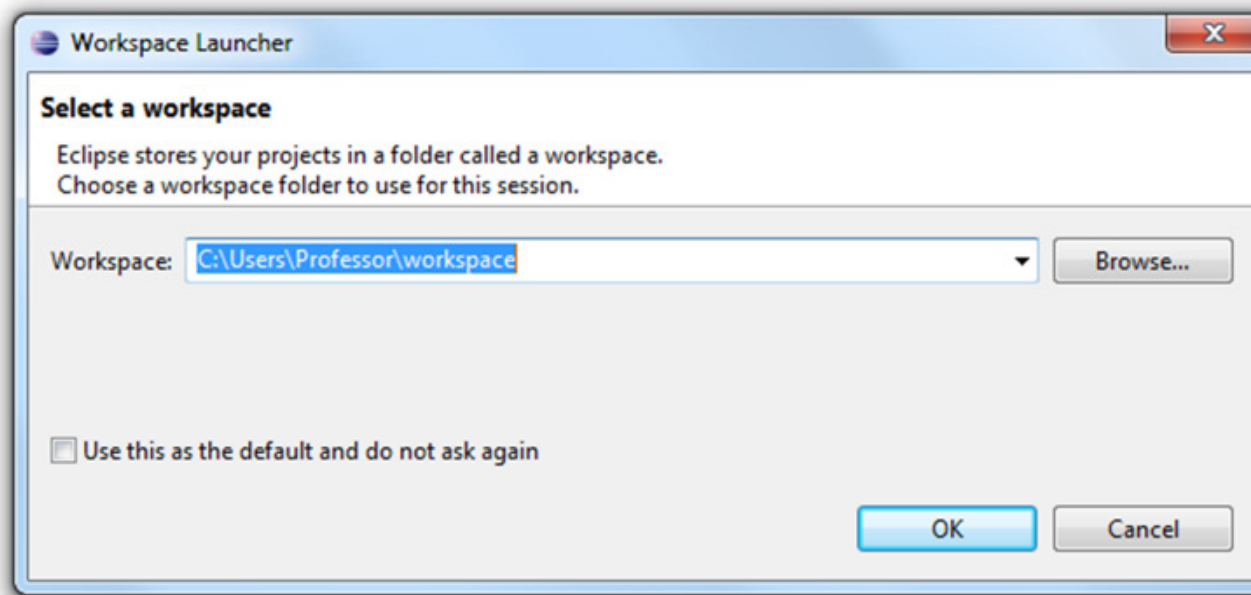
```
C:\JavaCode>java MadhavaLeibnizReihe  
Eine Naehierung fuer PI ist 3.140592653839794
```

Ein etwas geübter Programmierer erahnt einen solchen Fehler bereits beim Lesen des Codes. Das strategisch günstige Ausgeben von Zwischenergebnissen und Variablenwerten ist jedoch auch ein probates Mittel zum Aufspüren von schwierigeren Fehlern.

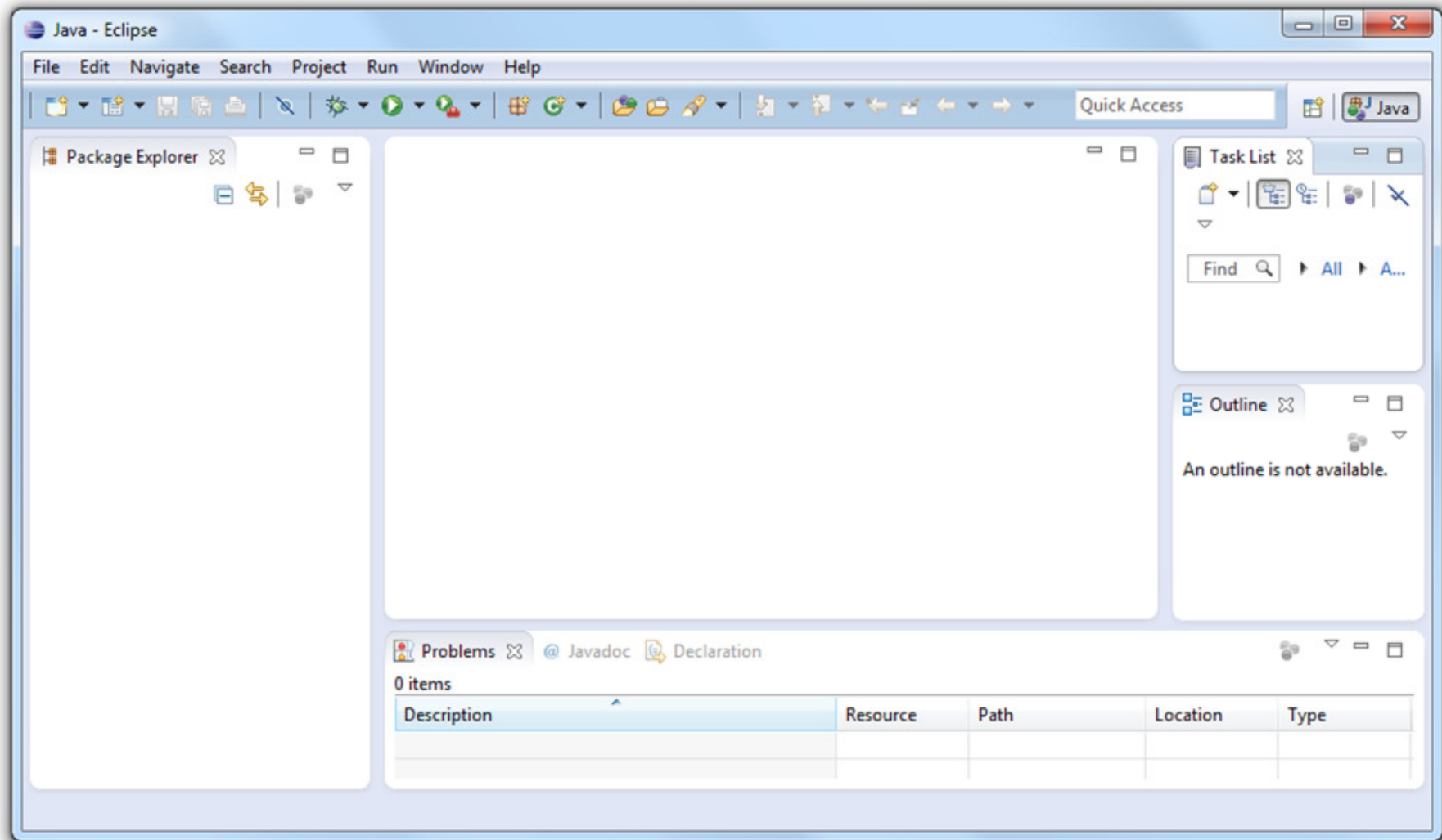
Wird das Programm jedoch größer, ist Code von anderen Programmieren beteiligt und sind die Berechnungen unübersichtlich, dann helfen die von den meisten Programmierumgebungen bereitgestellten Debugger enorm. Die meisten Systeme erlauben Breakpoints (Haltepunkte), bei denen alle aktuell im Speicher befindlichen Variablen und Zustände ausgelesen werden können.

Entwicklungsumgebung Eclipse - Workspace auswählen

Installieren, erster Start, erstes Projekt anlegen und Klasse starten



Entwicklungsumgebung Eclipse



Testgetriebene Softwareentwicklung

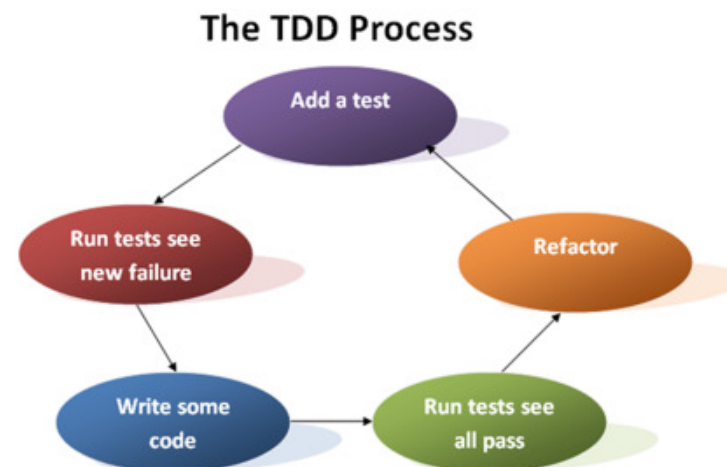
Die vollständige Verifikation von Programmen kann teilweise sehr schwierig oder gar unmöglich sein und ist oft mit einem sehr hohen Zeitaufwand verbunden. Der Wunsch nach kürzeren Entwicklungszeiten und die Notwendigkeit, dabei möglichst fehlerfreie Programme zu entwickeln, führte zu einem neuen Vorgehensmodell: die testgetriebene Entwicklung (engl. test-driven development oder kurz TDD).

TDD wird heutzutage oft in der agilen Softwareentwicklung eingesetzt. Im Gegensatz zu klassischen Vorgehensmodellen, wie dem Wasserfallmodell, V-Modell, Spiralmodell oder weiteren, wird bei der agilen Softwareentwicklung versucht, mit einem möglichst geringen protokolarischem Aufwand, weniger Regeln, wenig Dokumentationsaufwand und iterativen Schritten vorzugehen.

Dieses Vorgehensmodell wird heutzutage oft bei der Entwicklung von Computerspielen (speziell mit Scrum) eingesetzt, findet aber auch bei der Entwicklung von sicherheitskritischen Systemen wie der Automobilindustrie Anwendung.

Das typische Vorgehen

Bei der testgetriebenen Entwicklung wird vor der eigentlichen Implementierung einer Methode das gewünschte, fehlerfreie Verhalten in einem Testszenario beschrieben. Auf diese Weise erhalten wir eine ausführbare Spezifikation. Diese dokumentiert neben der Funktionalität gleich die Fehlerfreiheit des Systems. Sollten im späteren Programmverlauf weitere Fehler auftreten, an die zuvor nicht gedacht wurde, werden die Tests an dieser Stelle einfach erweitert, um zukünftig auch diese Fälle abzudecken. Sollten die Tests jetzt gestartet werden, so besteht die Methode diese selbstverständlich nicht.



Im zweiten Schritt wird die Methode mit minimalem Aufwand implementiert, bis die Tests erfolgreich verlaufen.

Anschließend wird die Methode schrittweise aufgeräumt (engl. *refactoring*). Dabei muss diese Methode die Tests selbstverständlich weiterhin bestehen.

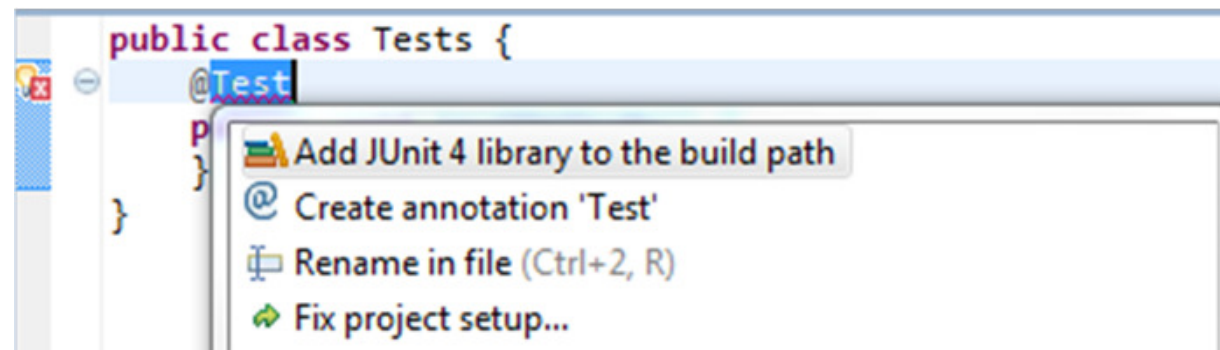
JUnit in Eclipse

Das Framework JUnit, das bereits in Eclipse vorhanden und leicht anzuwenden ist, bietet uns einen hervorragenden Einstiegspunkt in TDD.

Zunächst einmal müssen wir dem Projekt mitteilen, dass wir das aktuelle JUnit verwenden wollen. Dazu legen wir jetzt einen Ordner tests in kapitel04 und dort eine Klasse Tests mit folgendem Inhalt an:

```
public class Tests {  
    @Test  
    public void testMethod() {}  
}
```

Wenn wir folgende Methode mit dem Zusatz @Test einfügen, wird dieser Zusatz als Fehler angezeigt. Mit einem kurzen Verweilen der Maus auf dem Fehler oder der Tastenkombination <STRG+1> öffnet sich eine Liste mit Handlungsvorschlägen von Eclipse:



Der erste Test

Jetzt wollen wir die vorgeschlagene Musterlösung einer bereits bekannten Aufgabe untersuchen:

```
public static int[] addVektoren(int[] a, int[] b) {  
    if (a.length != b.length)  
        return null;  
    int[] sum = new int[a.length];  
    for (int i=0; i<a.length; i++)  
        sum[i] = a[i] + b[i];  
    return sum;  
}
```

Um einen verständlicheren Bezeichner für den bevorstehenden Test zu haben, nennen wir die Methode um und füllen sie mit folgendem Inhalt:

```
@Test  
public void testAddVektorenMitUnterschiedlichenLängen() {  
    int[] a = new int[]{1, 2, 3};  
    int[] b = new int[]{1, 2};  
    assertNull(Übung1.addVektoren(a, b));  
    assertNull(Übung1.addVektoren(b, a));  
}
```

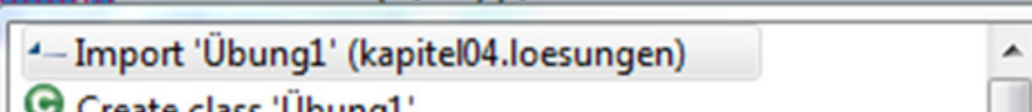
In diesem Fall wurde die Testmethode von keinem anderen Programmort verwendet. Wenn wir in Zukunft Bezeichner von Variablen, Methoden, Klassen usw. ändern wollen, machen wir das mit dem sogenannten Refactoring.

Dazu markieren wir den zu ändernden Bezeichner mit Doppelklick und drücken die Tastenkombination <ALT+SHIFT+R>, die auch im Eclipsemenü unter Refactor/Rename zu finden ist. Jetzt können wir den neuen Bezeichner eingeben und das Ganze mit Return abschließen.

Der erste Test II

Wir sehen, dass Übung1 rot markiert ist, denn die Klasse ist hier noch nicht bekannt. Mit einem Doppelklick auf Übung1, was den Namen komplett markiert, und der Tastenkombination <STRG+1> werden uns Vorschläge :

```
@Test
public void testAddVektorenMitUnterschiedlichenLängen() {
    int[] a = new int[]{1, 2, 3};
    int[] b = new int[]{1, 2};
    assertNull(Übung1.addVektoren(a, b));
    assertNull(
}
```



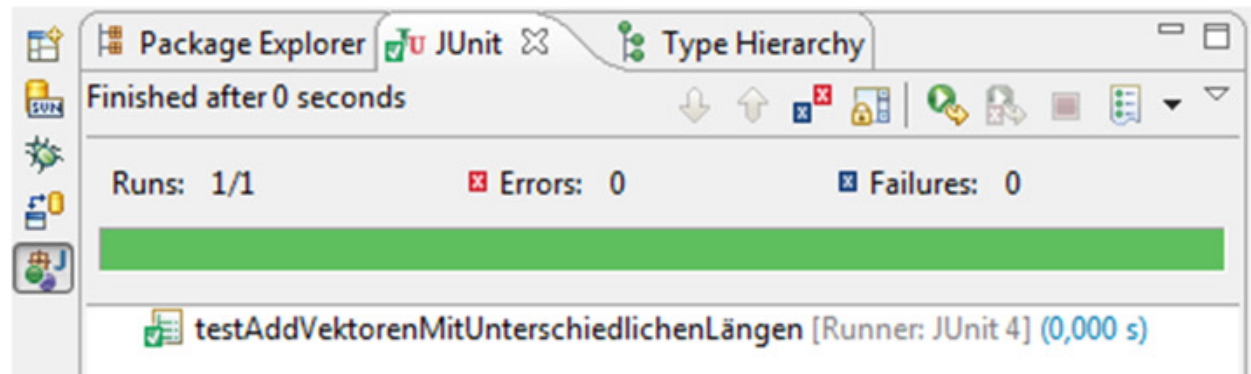
Wir wählen Import 'Übung 1' (kapitel04.loesungen). Eclipse unterstützt uns bei der Entwicklung von Programmen und hilft bei der Fehleranalyse. Auch assertNull ist rot markiert und kann auf die gleiche Weise wie zuvor korrigiert werden. In diesem Fall wählen wir Add static import 'org.junit.Assert.assertNull'.

Was macht nun eigentlich der Test? Der Bezeichner verrät, dass die Methode addVektoren mit Vektoren unterschiedlicher Länge getestet werden soll. Dazu werden zwei Listen a und b angelegt.

Der erste Test III

Das Ergebnis des Aufrufs `Übung1.addVektoren(a, b)` wird an die Methode `assertNull` übergeben. Diese besteht einen Test nur, wenn die Eingabe null ist.

Ein Doppelklick auf den Bezeichner der Methode und dann `Run As/JUnit Test` startet den Test. Im JUnit-View, einem Bereich von Eclipse, zeigt sich der erfolgreiche Test:



So einfach war der erste Test.

Der zweite Test

Jetzt wollen wir testen, ob die Methode `addVektoren` zwei Vektoren komponentenweise korrekt addiert:

```
@Test
public void testAddVektorenKomponentenweise() {
    int[] a = new int[]{1, 2, 3};
    int[] b = new int[]{1, 2, 4};
    int[] erg = new int[]{2, 4, 7};
    int[] berechnet = Übung1.addVektoren(a, b);
    assertEquals(erg, berechnet);
    berechnet = Übung1.addVektoren(b, a);
    assertEquals(erg, berechnet);
}
```

Die Methode `assertEquals` überprüft elementweise die einzelnen Inhalte zweier gleichlanger Arrays und besteht nur, wenn diese gleich sind.

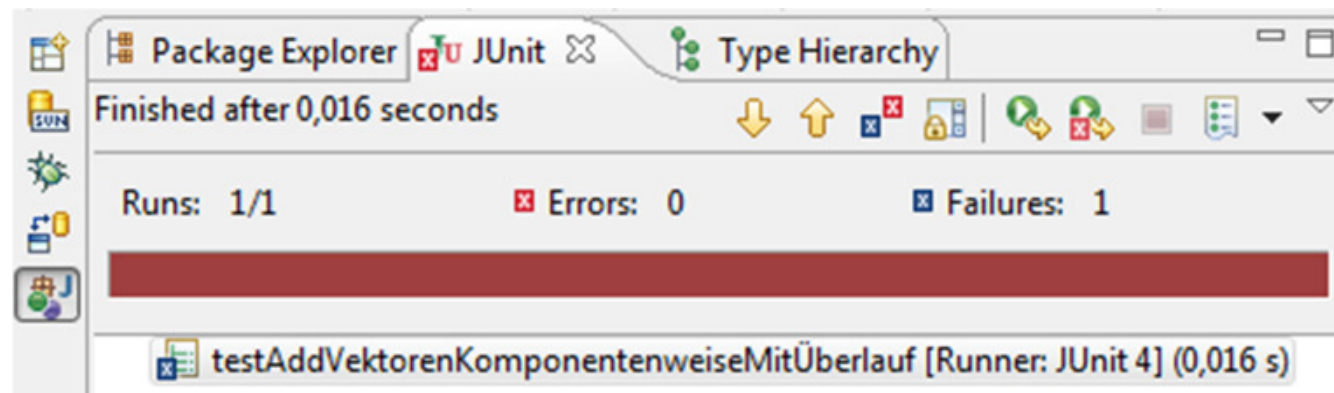
In diesem Beispiel sehen wir schon, dass das Testen von Eingaben zu den entsprechenden Ausgaben eine Verifikation nicht ersetzen kann, sondern lediglich eine kleine Stichprobe darstellt. Wir können also trotz umfangreicher Tests mit dieser Variante nie gewährleisten, dass das Programm hundertprozentig korrekt arbeitet. So könnten bestimmte Eingaben zu einem Speicherüberlauf führen und dann stimmt der Ergebnis nicht mehr.

Der zweite Test II

Jetzt sehen wir auch einen Speicherüberlauf als mögliche Fehlerquelle in unserer Methode `addVektoren`. Sollte es zu so einem Speicherüberlauf kommen, wollen wir jetzt, dass zukünftig `null` zurückgegeben wird. Wir schreiben allerdings zuerst einen Test, der das überprüft:

```
@Test
public void testAddVektorenKomponentenweiseMitÜberlauf() {
    int[] a = new int[]{Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE};
    int[] b = new int[]{1,1,1};
    int[] berechnet = Übung1.addVektoren(a, b);
    assertNull(berechnet);
    berechnet = Übung1.addVektoren(b, a);
    assertNull(berechnet);
}
```

Als Werte in `a` haben wir jeweils den größtmöglichen Wert für einen `int` und addieren eine 1 hinzu. Die Ausführung dieses Test führt zu einem Fehler:



Der zweite Test III

Jetzt können wir die Methode ändern zu:

```
public static int[] addVektoren(int[] a, int[] b) {  
    if (a.length != b.length)  
        return null;  
  
    int[] sum = new int[a.length];  
    for (int i=0; i<a.length; i++) {  
        if (a[i] >= Integer.MAX_VALUE - b[i])  
            return null;  
  
        sum[i] = a[i] + b[i];  
    }  
  
    return sum;  
}
```

Dabei wird vor jeder Addition überprüft, ob das Maximum größer oder gleich der Summe der beiden Werte ist. Ein weiterer Testlauf zeigt, dass das gewünschte Verhalten jetzt vorhanden ist. Oder haben wir etwas vergessen?

Was passiert eigentlich, wenn wir zwei sehr kleine Werte addieren, sagen wir Integer.MIN_Value und -1? Die Beantwortung dieser Frage wird Teil der Übungsaufgaben sein.

Übersicht zu den Annotationen von JUnit 4

Annotation	Beschreibung
<code>@Test</code>	Identifiziert eine Testmethode
<code>@Test (expected=Exception.class)</code>	Besteht den Test nur, wenn die beschriebene Exception geworfen wird
<code>@Test (timeout=100)</code>	Besteht den Test nicht, wenn die Methode mehr als 100 ms benötigt
<code>@BeforeClass</code>	Wird vor allen Tests einmal ausgeführt
<code>@Before</code>	Wird vor jedem Test ausgeführt
<code>@After</code>	Wird nach jedem Test ausgeführt
<code>@AfterClass</code>	Wird nach allen Tests einmal ausgeführt
<code>@Ignore</code>	Der Test wird nicht ausgeführt

Übersicht zu den Testmethoden von JUnit 4

Testmethode	Beschreibung
<code>fail(String)</code>	Todo
<code>assertTrue(boolean)</code>	Prüft, ob der boolean true ist
<code>assertFalse(boolean)</code>	Prüft, ob der boolean false ist
<code>assertNull(<Typ>)</code>	Prüft, ob die Eingabe null ist
<code>assertNotNull(<Typ>)</code>	Prüft, ob die Eingabe ungleich null ist
<code>assertEquals(double, double, double)</code>	Prüft, ob die ersten beiden double mit der Toleranz im dritten Parameter gleich sind
<code>assertEquals(<Typ>, <Typ>)</code>	Prüft, ob die Inhalte beider Parameter gleich sind
<code>assertArrayEquals(<Typ> [], <Typ> [])</code>	Prüft, ob der Inhalt beider Listen elementweise gleich ist
<code>assertSame(<Typ>, <Typ>)</code>	Prüft, ob beide Eingaben auf dasselbe Objekt referenzieren
<code>assertNotSame(<Typ>, <Typ>)</code>	Prüft, ob beide Eingaben auf unterschiedliche Objekte referenzieren

Refactoring

Da die deutschen Bedeutungen Umgestaltung oder Restrukturierung in dem Kontext der testgetriebenen Entwicklung selten auftauchen und stattdessen Refactoring verwendet wird, wollen wir das ebenso machen.

Mit **Refactoring** meinen wir die Verbesserung von Programmen z.B. durch Umbenennungen von Variablen, Auslagern von Methoden, Umpositionieren von Code ohne die eigentliche Funktionalität des Programms zu verändern -- eben alles, was der besseren Lesbarkeit und Verständlichkeit eines Programms dient.

Vom Refactoring sollte immer dann Gebrauch gemacht werden, wenn es den eigenen und fremden Code lesbarer und verständlicher macht. Hier gilt das oft verwendete Pfadfindermotto von Robert Baden-Powell:

Hinterlasse einen Ort sauberer als Du ihn vorgefunden hast.



Jetzt werden wir sehen, wie uns Eclipse bei der Verbesserung von Programmen unterstützt. Alle hier vorgestellten Tastenkombinationen lassen sich ebenfalls die Verwendung des Menüs Refactor ersetzen.

Bezeichner umbenennen

Das einfache Umbenennen einer Methode oder eine Variable kann Änderungen an sehr vielen anderen Orten nach sich ziehen. Daher ist es sinnvoll, Umbenennungen durch Eclipse vornehmen zu lassen. Dazu schauen wir uns nochmal das Fakultätsbeispiel mit Exception an.

```
public class FakultaetException {
    public static long fakultaet(int n) throws Exception {
        if ((n < 0) || (n > 20))
            throw new Exception("AUSSERHALB DES ZULAESSIGEN BEREICHS");
        long erg = 1;
        for (int i = n; i > 1; i--)
            erg *= i;
        return erg;
    }

    public static void main(String[] args) {
        for (int i = -2; i < 22; i++) {
            try {
                System.out.println("Fakultaet von "+i+" ist "+fakultaet(i));
            } catch (Exception e) { System.out.println(e.getMessage()); }
        }
    }
}
```

Um beispielsweise die Variable `erg` in produkt umzubennenen, markieren wir `erg` gefolgt von der Tastenkombination `<ALT+SHIFT+R>`. Jetzt können wir den Bezeichner verändern und schließen die Änderung mit `<RETURN>` ab. Sollten wir bei der Umbenennung unglücklicherweise einen bereits vorhandenen Bezeichner erwischen, in diesem Fall z. B. `i`, dann warnt Eclipse davor und wir können den Vorgang abbrechen.

Programmcode in Funktionen auslagern

Dazu schauen wir uns noch einmal ein früheres Programmbeispiel an, das uns damals motivieren sollte, Programmcode in Funktionen auszulagern:

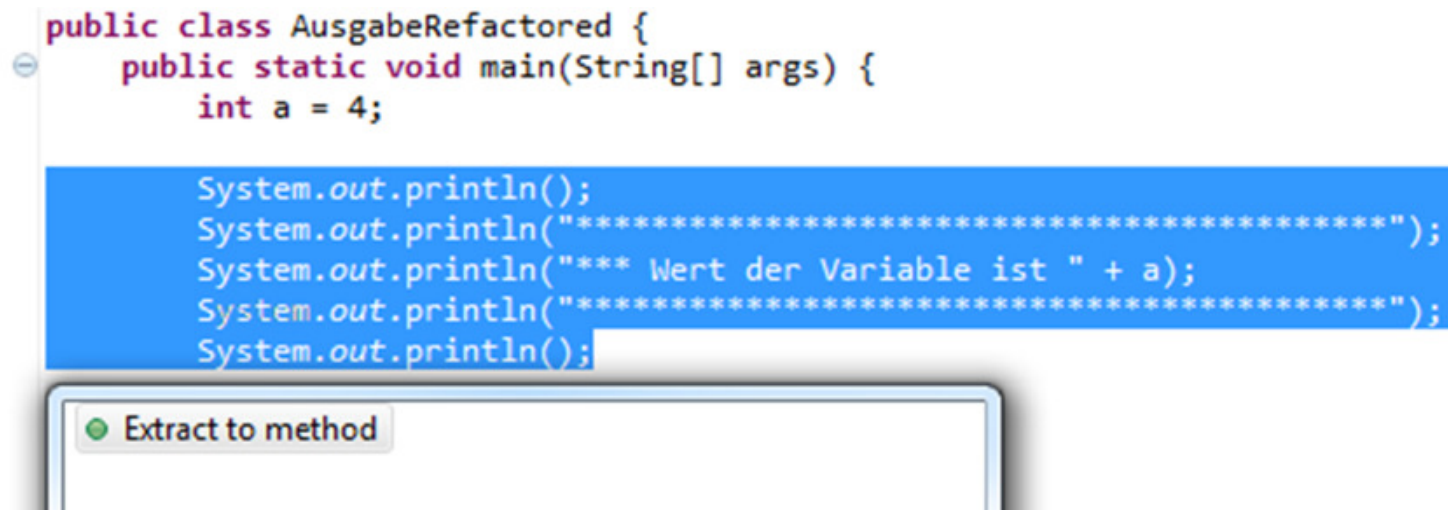
```
public class Ausgabe{
    public static void main(String[] args){
        int a=4;
        System.out.println();
        System.out.println("*****");
        System.out.println("*** Wert der Variable ist "+a);
        System.out.println("*****");
        System.out.println();

        a=(a*13)%12;
        System.out.println();
        System.out.println("*****");
        System.out.println("*** Wert der Variable ist "+a);
        System.out.println("*****");
        System.out.println();

        a+=1000;
        System.out.println();
        System.out.println("*****");
        System.out.println("*** Wert der Variable ist "+a);
        System.out.println("*****");
        System.out.println();
    }
}
```

Programmcode in Funktionen auslagern

Wir sehen gleich, dass es redundante Programmzeilen gibt. Um diese in eine eigene Funktion auszulagern, markieren wir das erste Auftreten der Ausgabezeilen und holen uns mit <STRG+1> Unterstützung vom Eclipse-Ratgeber. Dieser wird Extract to method anbieten:



Programmcode in Funktionen auslagern II

Wenn wir den Vorschlag anklicken, werden alle drei identischen Programmabschnitte erkannt und in eine Funktion ausgelagert. Der erste Eintrag bleibt markiert, was bedeutet, dass wir dem extrahierten Programmabschnitt einen passenderen Bezeichner geben können:

```
public class AusgabeRefactored {  
    public static void main(String[] args) {  
        int a = 4;  
  
        extracted(a);  
  
        a = (a * 13) % 12;  
  
        extracted(a);  
  
        a += 1000;  
  
        extracted(a);  
    }  
  
    private static void extracted(int a) {  
        System.out.println();  
        System.out.println("*****");  
        System.out.println("*** Wert der Variable ist " + a);  
        System.out.println("*****");  
        System.out.println();  
    }  
}
```

Programmcod in Funktionen auslagern III

Das nächste Mal können wir für das Extrahieren auch die Tastenkombination `<ALT+SHIFT+M>` verwenden.

Mit diesem Schritt haben wir nicht nur redundanten Programmcod vermieden, sondern Funktionalität für eine bessere Wartbarkeit gekapselt. Eclipse erkennt sogar automatisch, welche Parameter übergeben werden und was die Funktion zurückliefert.

Wie wir ja bereits wissen, liefert eine Funktion immer genau einen Rückgabewert. Daher klappt das Extrahieren an den Stellen nicht, wo mehrere, potentielle Rückgabewerte vorliegen.

Variablen extrahieren

Wir haben uns bereits ein wenig mit der Kreiszahl pi beschäftigt. Die 1995 veröffentlichte Bailey-Borwein-Plouffe-Formel ist ebenfalls eine Reihe zur Bestimmung von pi, hat unter anderem den Vorteil, sehr viel schneller zu konvergieren:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

Diese Formel wurde hier implementiert, liefert aber leider für pi mit 10 Iterationsschritten den Wert 2.0644030597309695:

```
public static double piBailey(int maxIterationen) {  
    double pi = 0;  
    int currI = 0;  
    while (currI < maxIterationen) {  
        pi += 1.0/Math.pow(16, currI)*4.0/(8*currI+1)  
            -2.0/(8*currI+4)-1.0/(8*currI+5)-1.0/(8*currI+6);  
        currI++;  
    }  
    return pi;  
}
```

Variablen extrahieren II

Eigentlich sieht die Implementierung der Formel korrekt aus. Trotzdem ist sie sehr lang und unübersichtlich. Mit Hilfe der Variablenextraktion wollen wir ein wenig Übersicht verschaffen und die beiden multiplikativen Terme der Summe zerlegen.

Dazu markieren wir den zweiten langen Term gefolgt von der Tastenkombination <STRG+1> und wählen Extract to local variable (alternativ können wir auch <ALT+SHIFT+L> verwenden):

```
double d = 4.0;  
pi += 1.0/Math.pow(16, currI)*d/(8*currI+1)-2.0/(8*currI+4)  
      -1.0/(8*currI+5)-1.0/(8*currI+6);
```

Wir erinnern uns, dass Refactoring die vorliegende Funktionalität nicht ändern darf, sondern lediglich die Lesbarkeit erhöhen soll. Scheinbar drückt unser erster Implementierungsversuch nicht ganz die Formel aus.

Variablen extrahieren III

Versuchen wir den Term vor dem Extrahieren mal zu klammern. Wir erhalten anschließend:

```
double term = 4.0/(8*currI+1)-2.0/(8*currI+4)-1.0/(8*currI+5)-1.0/(8*currI+6);  
pi += 1.0/Math.pow(16, currI) * term;
```

Jetzt sehen wir auch, was für falsch gemacht haben. Die Klammern waren hier absolut notwendig!

Wenn wir das Programm erneut mit 10 Iterationen starten, erhalten wir den Wert 3.1415926535897913 , also pi auf 14 Nachkommastellen genau.

Programmcode formatieren

Durch die vergangenen Codebeispiele wurde klar, wie wichtig es ist, den Programmcode nicht nur verständlich zu implementieren sondern auch vernünftig zu formatieren. Dabei war es bisher jedem selbst überlassen, ob das Einrücken einer Programmzeile durch zwei, drei oder noch mehr Leerzeichen bzw. eine Tabulatoreinrückung realisiert wird. Wenn allerdings viele Leute zusammenarbeiten, dann ist es sinnvoll, eine gemeinsame Formatierung zu verwenden.

Eclipse formatiert den Programmcode automatisch, wenn wir ihn markieren und die Tastenkombination `<STRG+SHIFT+F>` verwenden. Sollte das Team entscheiden, die Formatierung für ein Projekt anzupassen, so ist das problemlos möglich, z. B. über `Project/Properties/Java Code Style/Formatter`.

Übersicht zu wichtigen Tastenkombinationen in Eclipse

Tastenkombination	Beschreibung
<STRG+1>	Kontextsensitive Vorschläge zur Programmkorrektur oder Autovervollständigung
<STRG+LEERTASTE>	Autovervollständigung
<SHIFT+F2>	Kontexthilfe der API-Dokumentation
<STRG+SHIFT+F>	Markierten Programmabschnitt formatieren
<ALT+SHIFT+R>	Bezeichner umbenennen
<ALT+SHIFT+M>	Markierten Programmabschnitt in Methode extrahieren
<STRG+SHIFT+7>	Markierten Programmabschnitt aus- bzw. einkommentieren
<STRG+SHIFT+O>	Organisiert die imports: Entfernt überflüssige, fügt fehlende hinzu
<ALT+SHIFT+S>	Kontextmenü für Programmcodeaktionen, wie z. B. automatische Erzeugung von get- und set-Methoden

Hinweise für testgetriebene Entwicklung

Wir beginnen mit den drei Regeln testgetriebener Entwicklung:

- 1.** Schreibe **erst** einen **Modultest** der fehlschlägt und **dann** den dazugehörigen **Programmcode**.

Original: *You are not allowed to write any production code until you have first written a failing unit test.*



- 2.** Mehr Aufwand als notwendig bei der Erstellung zum Scheitern eines Modultests ist nicht erlaubt.

Original: *You are not allowed to write more of a unit test than is sufficient to fail - and not compiling is failing.*



- 3.** Schreibe gerade soviel Programmcode, dass der aktuelle Modultest bestanden wird.

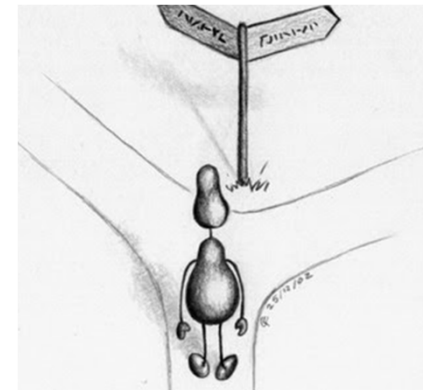
Original: *You are not allowed to write more production code that is sufficient to pass the currently failing unit test.*



Hinweise für testgetriebene Entwicklung II

Wir sehen in der Beschreibung der Regeln, den kontinuierlichen Zyklus des Refactorings. Schließlich entwickeln wir Testmethoden und Programmcode gleichzeitig. Daher müssen Tests mit der gleichen **Sorgfalt und Lesbarkeit** erstellt werden, wie der Programmcode. Spätere Änderungen im Programmcode ziehen auch Änderungen im Testcode nach sich. Wir halten damit den ganzen **Programmcode flexibel**, denn jede Änderung, die normalerweise Fehler nach sich ziehen könnte, wird von vielen Seiten durch die Test mit überwacht.

Bei der präzisen Überlegung, wie ein Test geschrieben wird, treffen wir bereits wichtige **Designentscheidungen**. Das kann uns helfen die Programmstruktur konkreter und wohlüberlegter zu gestalten.



Wie bei der Entwicklung von Funktionen gilt die Empfehlung, keine langen Testmethoden zu schreiben. Im besten Fall reicht ein Assertaufruf pro Methode. Robert C. Martin gibt uns mit dem Akronym **FIRST** (für Fast, Independent, Repeatable, Self-Validating und Timely) fünf weitere Empfehlungen für die Erstellung sauberer Tests.

Live-Coding-Session

```
    == "checked";  
    number++;  
    contents++;  
    $count; $1++;  
    $count type="|", $data/on  
    $i + 1, "|";  
    $i, $totalsecurity))  
    echo "checked";  
    if ($i == 0) {  
        ("checked");
```