4 Fehlersuche und Test

In diesem Kapitel geht es um das Finden von Fehlern. Auch wenn niemand gerne eingesteht, dass seine Programme sie enthalten, so ist es doch eine unumstößliche Tatsache, dass jede nicht triviale Software fehlerbehaftet ist. Deshalb gehört die Fehlersuche zu den grundlegenden Tätigkeiten eines Entwicklers. Entsprechend gut sollte das eingesetzte Werkzeug ihn hierbei unterstützen. Mit dem Begriff Debuggen verbinden viele instinktiv die Fehlersuche auf Quelltextebene. Gerade bei Compiler-Sprachen ist dies aber ein keineswegs selbstverständlicher Luxus. Denn bei der Übersetzung in Maschinensprache müssen Strukturen und Befehle auf eine viel niedrigere Abstraktionsebene gebracht werden. Zudem kann aus Sequenzen von Maschinenbefehlen keineswegs immer auf korrespondierende hochsprachliche Elemente geschlossen werden. Vereinfacht ausgedrückt, dem Debugger muss der Quelltext des zu untersuchenden Programms vorliegen, damit er den aktuellen Zustand während der Programmausführung auf diesen abbilden kann. So war die Ausgabe von Variableninhalten oder Positionsangaben durch `print()`-Funktionen oder -Methoden lange Zeit das wichtigste Hilfsmittel bei der Fehlersuche. Und sie wird selbst heutzutage noch gerne eingesetzt.

Im Folgenden möchte ich Ihnen zeigen, wie Ihnen Eclipse beim Aufspüren von Programmfehlern hilft. In Abschnitt 4.1, „Visuelles Debuggen“, lernen Sie die Perspektive *Debug* mit ihren zahlreichen Sichten kennen. Der Abschnitt 4.2, „Konzepte und Vorgehensweisen“, zeigt Ihnen unter anderem, wie Sie während des Programmablaufs Variableninhalte einsehen und Breakpoints setzen. In Abschnitt 4.3, „Fortgeschrittene Debug-Techniken“, lernen Sie weiterführende Aspekte der Fehlersuche kennen, beispielsweise bedingte Breakpoints sowie ausgefeilte Techniken zur Einzelschrittsteuerung. Und getreu dem Motto „Am leichtesten lassen sich nicht gemachte Fehler ausbügeln“ stelle ich Ihnen schließlich in Abschnitt 4.4, „Unit-Tests“, Konzepte vor, mit denen Sie potenzielle Probleme schon während der Programmierung entdecken können.

4.1 Visuelles Debuggen

Visuelles oder interaktives Debuggen ist, wenn Sie so wollen, die Königsdisziplin der Fehlersuche. Der Entwickler sieht während der Programmausführung seinen Quelltext und kann praktisch jederzeit die Abarbeitung anhalten, Variablen einsehen und verändern.

Sogar Modifikationen am Programm sind möglich. Allerdings geht mit dieser Flexibilität eine gewisse Einarbeitungszeit einher. Oder positiver formuliert: Es gibt viel zu entdecken und auszuprobieren.

4.1.1 Ein erstes Beispiel

Eclipse bettet die Fehlersuche in eine eigene Perspektive ein. Um diese kennenzulernen, stelle ich Ihnen ein kleines Programm vor, das die Fakultät einer beliebigen natürlichen Zahl berechnet.

Fakultätsberechnung

Die Fakultät ist eine mathematische Funktion, die einer natürlichen Zahl das Produkt aller natürlichen Zahlen kleiner oder gleich dieser Zahl zuordnet. Sie wird durch ein Ausrufezeichen, das dieser Zahl nachgestellt ist, abgekürzt. Als natürliche Zahlen bezeichnet man alle positiven ganzen Zahlen. Die Null wird üblicherweise nicht als natürliche Zahl betrachtet. Es gilt also beispielsweise 5! = 5 × 4 × 3 × 2 × 1. Fakultäten für negative oder für nicht ganze Zahlen sind nicht definiert. Gemäß Definition ist 0! = 1. Aus diesen Vorgaben lässt sich sehr leicht ein kurzes Programm ableiten. Es berechnet die Fakultät einer Zahl aus dem Produkt der Fakultät ihres Vorgängers und der Zahl selbst.

Listing 4.1 Die Klasse Fakultaet

```
package com.thomaskuenneth.eclipse_book;

public class Fakultaet {

  public static int fakultaet(int zahl) {
    int ergebnis = 1;
    if (zahl > 1) {
      ergebnis = zahl * fakultaet(zahl - 1);
    }
    return ergebnis;
  }

  public static void main(String[] args) {
    if (args.length < 1) {
      System.exit(1);
    }
    int zahl = Integer.parseInt(args[0]);
    int ergebnis = fakultaet(zahl);
    System.out.print(zahl + "! = " + ergebnis);
  }
}
```

Die Methode `fakultaet()` ruft sich so lange selbst auf, bis die übergebene Zahl kleiner als 2 ist. Dann nämlich ist das Ergebnis ohne weitere Rekursion ermittelbar. Denn es gilt 0! = 1! = 1. Falls Sie Zweifel haben, dass diese geschachtelten Aufrufe tatsächlich nötig sind, haben Sie natürlich recht. Die Multiplikationen könnten Sie viel leichter und effizienter in Form einer Schleife realisieren. Allerdings lassen sich mit rekursiven Methodenaufrufen sehr schön grundlegende Techniken der Fehlersuche demonstrieren. In den folgenden Abschnitten werden Sie die wichtigsten Werkzeuge kennenlernen, die Eclipse für die

Fehlersuche anbietet. Dabei lasse ich bewusst einige Einstellmöglichkeiten und Funktionen außen vor, um Ihnen zunächst ein Gesamtbild zu vermitteln. Später geht es dann mit fortgeschrittenen Techniken der Fehlersuche weiter.

Debug-Vorgang starten

In Kapitel 2, „Arbeiten mit Eclipse“, habe ich Ihnen gezeigt, wie Programme gestartet werden. Der Aufruf des Debuggers geschieht auf sehr ähnliche Weise. Beispielsweise können Sie eine Fehlersuche beginnen, indem Sie **Run/Debug** oder **Run/Debug As/Java Application** auswählen. Klicken Sie stattdessen aber bitte auf **Run/Debug Configurations**. Sie sehen den Ihnen bereits bekannten Dialog zum Anlegen, Bearbeiten und Löschen von Konfigurationen. Um eine neue Debug-Konfiguration anzulegen, klicken Sie mit der rechten Maustaste auf **Java Application** und wählen Sie dann **New Configuration**. Auf der Registerkarte **Main** selektieren Sie das Projekt, in dem Sie die Klasse `Fakultaet` angelegt haben, und geben `com.thomaskuenneth.eclipse_book.Fakultaet` als **Main class** an. Setzen Sie außerdem ein Häkchen vor **Stop in main**. Wechseln Sie nun auf die Registerkarte **Arguments** und klicken Sie unter **Program arguments** auf **Variables**. Sie sehen daraufhin den in Bild 4.1 gezeigten Dialog *Select Variable*. Er enthält eine Liste mit Variablen, die Sie Programmen beim Start übergeben können. Mit Hilfe der Eingabezeile **Argument** steuern Sie, wie der Wert einer Variablen ermittelt wird.

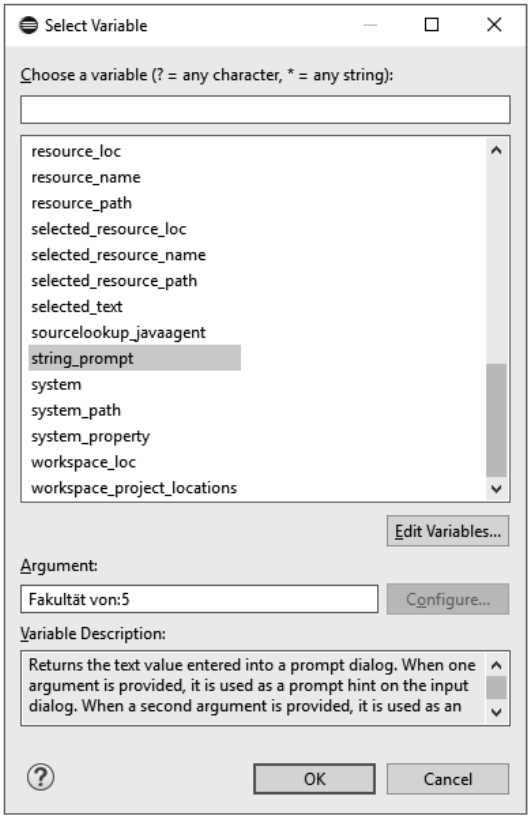

Bild 4.1 Der Dialog „Select Variable“

Beispielsweise erwartet **system_property** den Namen einer System-Property wie `user.home`. Die Variable **string_prompt** sorgt dafür, dass Eclipse beim Start eines Programms einen kleinen Eingabedialog öffnet. Diesen können Sie in sehr begrenztem Umfang konfigurieren. Der erste Wert bei Argument enthält einen Hinweistext. Der zweite wird als Vorbelegung der Eingabezeile genutzt. Ein Doppelpunkt trennt die beiden Parameter. Nachdem Sie **string_prompt** ausgewählt haben, schließen Sie den Dialog *Select Variable* mit **OK**. Änderungen an Ihrer Debug-Konfiguration übernehmen Sie mit **Apply**. Nun können Sie den Debug-Vorgang starten. Klicken Sie hierzu auf **Debug**.

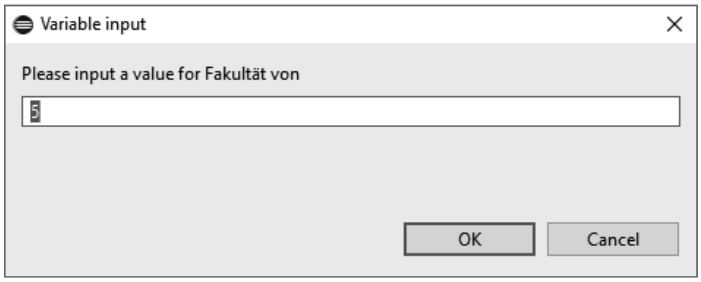

Bild 4.2 Der Dialog „Variable input“

Sie sehen den in Bild 4.2 gezeigten Dialog *Variable input*, in den Sie die zu berechnende Fakultät eintragen. Nachdem Sie die Eingabe mit **OK** bestätigt haben, informiert Sie Eclipse darüber, dass die IDE automatisch die Perspektive *Debug* öffnen kann. Sie sollten dem zustimmen und diese Einstellung speichern. Setzen Sie hierzu ein Häkchen vor **Remember my decision** und klicken Sie dann auf **Switch**. Eclipse wechselt daraufhin zur Perspektive *Debug*. Sie ist in Bild 4.3 zu sehen.

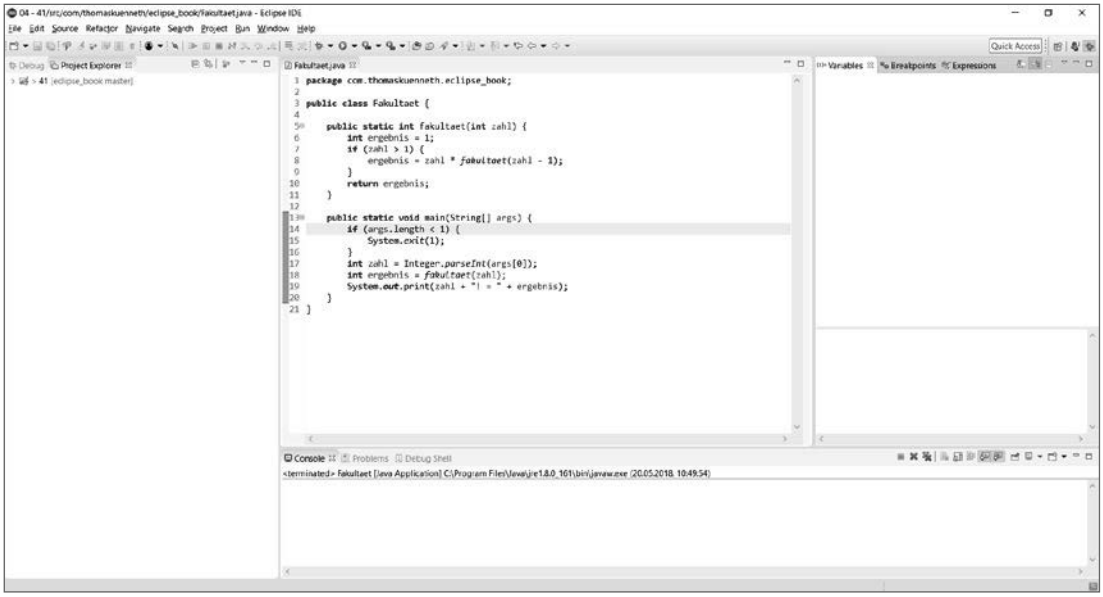

Bild 4.3 Die Perspektive „Debug“

Ein Element des Darstellungsbereichs ist der Java-Editor, der den Quelltext der Klasse `Fakultaet` enthält. Eine Zeile ist (in den Standardfarbeinstellungen) grün hinterlegt und zeigt im Randbereich durch einen kleinen blauen Pfeil die nächste auszuführende Anweisung an. Da Sie Eclipse angewiesen hatten, in `main()` anzuhalten, ist dies die erste Zeile dieser Methode.

Das Menü **Run** enthält in der Perspektive *Debug* zahlreiche Befehle, die die Abarbeitung eines Programms während einer Debug-Sitzung steuern. Beispielsweise können Sie mit **Resume** ein angehaltenes Programm weiterlaufen lassen. **Terminate** beendet es. Die Funktion **Step Over** führt die Anweisungen in der aktuellen Zeile aus. Die Abarbeitung wird vor dem ersten Befehl der Folgezeile wieder gestoppt. Probieren Sie dies aus, indem Sie einmal die Taste **F6** drücken. Erwartungsgemäß wird die nächste Zeile markiert. Wenn dem Programm über die Kommandozeile Argumente übergeben wurden, ist dies `int zahl = Integer.parseInt(args[0]);`. Drücken Sie erneut **F6**. Die nun markierte Zeile enthält einen Aufruf der Methode `fakultaet()`. Nach einem weiteren **Run/Step Over** stoppt die Abarbeitung vor der letzten Programmzeile, der Ausgabe des Ergebnisses durch **print()**. Führen Sie diese Zeile nun aus, indem Sie auf **Run/Resume** klicken. Das Ergebnis der Berechnung wird in der Sicht *Console* ausgegeben. Der Befehl Step Over erlaubt also die zeilenweise Ausführung eines Programms. Um die Berechnung der Fakultät verfolgen zu können, ist allerdings ein anderes Vorgehen nötig. Denn bei Methodenaufrufen findet keine Einzelschrittverarbeitung statt. Step Over behandelt Methoden, als wären sie einzelne Anweisungen.

Methodenaufrufe verfolgen

Starten Sie den Debug-Vorgang erneut, indem Sie **Run/Debug** aufrufen oder die Taste **F11** (auf dem Mac zusammen mit der **Command**-Taste) drücken. Geben Sie im Dialog *Variable input* den Wert „2“ ein. Führen Sie nun die `if`-Abfrage der Feldlänge sowie die Zuweisung der in eine Zahl gewandelten Eingabe jeweils mit **Step Over** aus. Zum Verfolgen eines Methodenaufrufs steht der Befehl **Run/Step Into** zur Verfügung. Sie erreichen ihn auch über das Tastenkürzel **F5**. Drücken Sie nun diese Taste. Eclipse wird daraufhin die erste Zeile des Rumpfs der Methode `fakultaet()` grün hinterlegen. Drücken Sie nun zweimal die Taste **F6**. Da Sie die Fakultät von 2 berechnen möchten, ist die Bedingung `(zahl > 1)` erfüllt. Der Wert von `ergebnis` ist deshalb das Produkt aus 2 und dem Rückgabewert des Methodenaufrufs `fakultaet(1)`. Um zu erfahren, wie sich `fakultaet()` bei 1 verhält, wählen Sie **Run/Step Into**. Ab jetzt können Sie den weiteren Programmverlauf so lange durch Drücken von **F6** verfolgen, bis Sie das Ende von `main()` erreichen. Rufen Sie dann **Resume** auf. **Step Into** löst also wie **Step Over** eine Einzelschrittverarbeitung aus, betrachtet Methodenaufrufe aber nicht als Blackbox, sondern stoppt die Verarbeitung vor der ersten Zeile des Methodenrumpfs. Das Menü Run enthält einige weitere Befehle, mit denen Sie die Programmausführung während einer Debug-Sitzung steuern können. Informationen hierzu finden Sie in Abschnitt 4.2, „Konzepte “.

4.1.2 Die Sichten der Perspektive „Debug“

Die Kontrolle des Programmflusses ist natürlich nur ein Aspekt bei der Fehlersuche. Um den Zustand eines Programms, beispielsweise Variableninhalte oder Aufrufabfolgen, zu visuali-

sieren, stellt die Perspektive *Debug* zahlreiche Sichten zur Verfügung. Falls diese nicht automatisch geöffnet werden, erreichen Sie die Sichten über **Window/Show View/Other** unterhalb des Knotens **Debug**. Der Dialog ist in Bild 4.4 zu sehen.

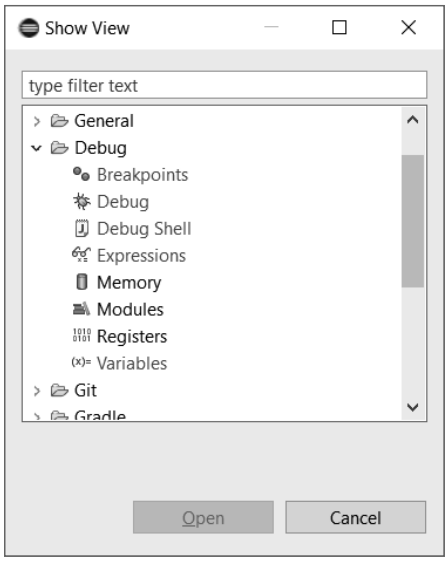

Bild 4.4 Der Dialog „Show View“

Die Sicht „ Debug“

Die in Bild 4.5 dargestellte Sicht fungiert als Steuerzentrale während einer Debug-Sitzung. Sie ordnet die Threads einer Anwendung in einer baumartigen Struktur an. Deren Wurzel repräsentiert die Debug-Konfiguration, die der aktuellen Sitzung zugrunde liegt. Weitere Elemente symbolisieren den Client- und Serverteil des Debuggers. Was es hiermit auf sich hat, erfahren Sie in Abschnitt 4.2.1, „Architektur des Eclipse Debuggers“.

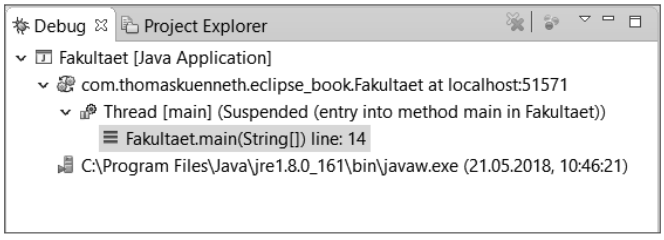

Bild 4.5 Die Sicht „Debug“

Ein Rechtsklick auf Elemente des Baums öffnet ein Kontextmenü, das neben den bereits bekannten Befehlen zur Programmsteuerung beispielsweise auch das Kopieren des Stacks auf das Systemklemmbrett oder das Bearbeiten der Debug-Konfiguration gestattet. Klicks mit der linken Maustaste auf die Elemente eines Thread-Knotens zeigen die korrespondierende Stelle im Quelltext an.

Die Symbolleiste der Sicht enthält einige Befehle zur Ablaufsteuerung, die Sie schon vom Menü **Run** her kennen, beispielsweise **Resume**, **Terminate**, **Step Into** und **Step Over**. Meine Empfehlung ist allerdings, solche gängigen Aktionen durch ihre entsprechenden Tastenkürzel auszulösen. Mit dem in Bild 4.6 gezeigten Klappmenü beeinflussen Sie die Anzeige der Sicht. Beispielsweise können Sie bei Bedarf System-Threads einblenden sowie Thread-Gruppen anzeigen. **Show Qualified Names** schaltet die Anzeige von Paketnamen (beispielsweise bei den Parametern von Methoden) ein oder aus.

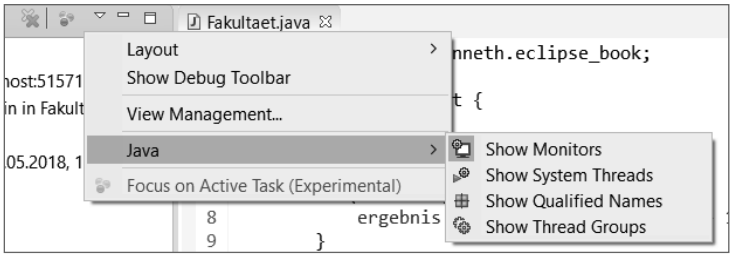

Bild 4.6 Klappmenü der Sicht „Debug“

Ferner können Sie über das Untermenü **Layout** die Ihnen bereits vom Java-Editor vertraute Brotkrümelnavigation steuern. Im Automatikmodus (**Automatic**) wird sie aktiviert, wenn die Sicht weniger als zwei Zeilen darstellen würde. Sie wird wie die Brotkrümelnavigation im Java-Editor bedient. Einzelne Elemente lassen sich anklicken, woraufhin sich ein Menü mit den möglichen Aktionen öffnet. Die Navigationsleiste ist über die Tastatur bedienbar.

Die Sicht „Debug Shell“

Mit der in Bild 4.7 gezeigten Sicht *Debug Shell* können Sie beliebige Ausdrücke auswerten. Die Sicht funktioniert wie das Scrapbook, das ich Ihnen in Kapitel 2, „Arbeiten mit Eclipse“, vorgestellt habe. Sie geben also zunächst den auszuwertenden Ausdruck ein und markieren ihn anschließend. Nun können Sie die gewünschte Funktion über die Symbolleiste oder das Kontextmenü der Sicht auslösen.

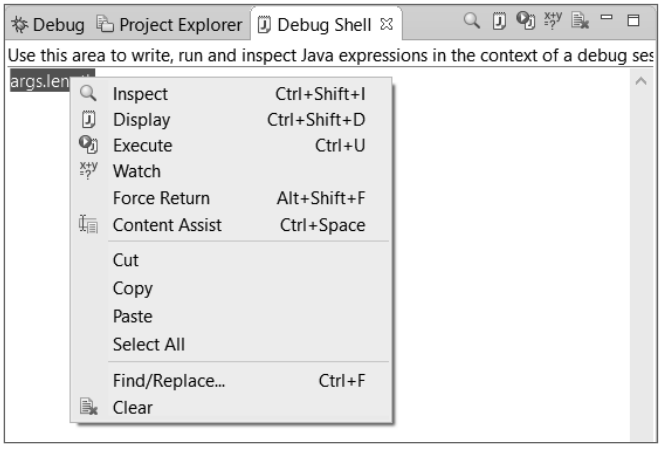

Bild 4.7 Die Sicht „Debug Shell“

Execute beispielsweise führt die Anweisungen aus, ohne das Ergebnis anzuzeigen. Dies ist praktisch, wenn Sie ohnehin Ausgaben in die Console machen, beispielsweise mit `System.out.println()`. **Display** hingegen zeigt das Ergebnis innerhalb der Sicht an. Wie bei Scrapbook kann dies dazu führen, dass der Text der Sicht keinen gültigen Java-Ausdruck mehr ergibt. Vor einer erneuten Auswertung müssen Sie ihn also gegebenenfalls editieren. Um dies zu verhindern, können Sie stattdessen die Funktion **Inspect** nutzen. Sie öffnet das in Bild 4.8 gezeigte Popup-Fenster. Es enthält das Ergebnis des ausgewerteten Ausdrucks. Mit dem Tastenkürzel **Strg + Shift + I** (auf dem Mac bitte Command drücken) lässt sich der Ausdruck übrigens in die Sicht *Expressions* übernehmen, die ich Ihnen im folgenden Abschnitt vorstelle.

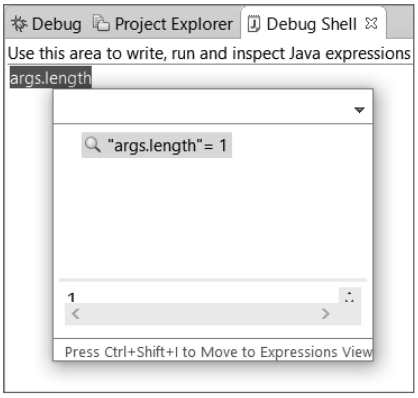

Bild 4.8 Popup mit dem Ergebnis eines ausgewerteten Ausdrucks

Um einen Ausdruck auszuwerten, müssen Sie ihn nicht erst in die Sicht *Debug Shell* übernehmen. Sie können ihn stattdessen direkt im Java-Editor markieren. Rufen Sie dann das Kontextmenü des Editors auf und wählen Sie **Display**. Das Ergebnis wird in einem Popup-Fenster angezeigt. Falls Sie den Ausdruck und sein Ergebnis in die Sicht *Debug Shell* kopieren möchten, drücken Sie **Strg** (bzw. **Command**) **+ Shift + D**, während das Ergebnis-Popup-Fenster geöffnet ist.

Die Sicht „Expre ssions“

Auch die in Bild 4.9 gezeigte Sicht *Expressions* stellt Ergebnisse von ausgewerteten Ausdrücken dar. Im Gegensatz zur Sicht *Debug Shell* beinhaltet sie aber keinen Scrapbook-ähnlichen Editor. Sie geben also die Ausdrücke nicht unmittelbar über die Sicht ein, sondern übernehmen sie beispielsweise aus der Sicht *Debug Shell* oder dem Java-Editor über die Funktion **Inspect**.

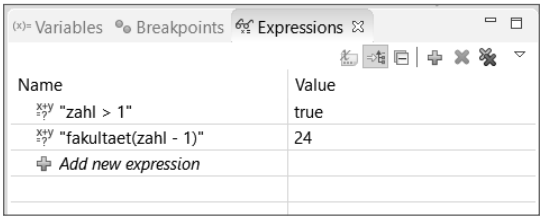

Bild 4.9 Die Sicht „Expressions“

Der Vorteil der Sicht liegt in einer sehr detaillierten Darstellung der ausgewerteten Ausdrücke. Außerdem können Sie gezielt auf Datenkonstellationen während einer Debug-Sitzung reagieren. Aus diesem Grund werden die hier dargestellten Ausdrücke Watch Expressions genannt.

Die Sicht „Variables“

Diese in Bild 4.10 gezeigte Sicht liefert eine übersichtliche (auf Wunsch tabellarische) Darstellung aller aktuell gültigen Variablen mit ihrem jeweiligen Inhalt. Über ein Kontextmenü lassen sich Variablen in die Sicht *Expressions* übernehmen, auf das Systemklemmbrett kopieren und Werte ändern. Sie können die Darstellung der Sicht über das Klappmenü in weiten Grenzen beeinflussen. Beispielsweise lassen sich Spaltenüberschriften sowie bestimmte Variablentypen ein- und ausblenden.

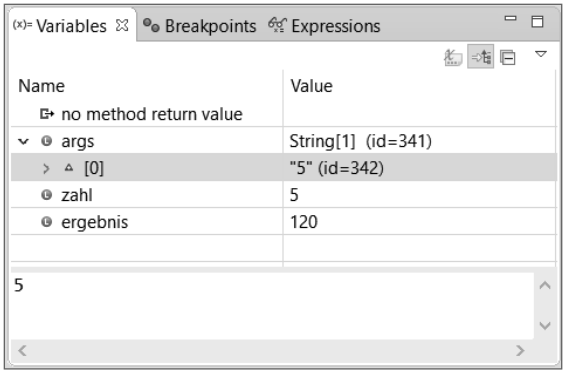

Bild 4.10 Die Sicht „Variables“

Die Sicht gestattet einen umfassenden Einblick in den Zustand eines Programms. Besonders interessant ist, dass Werte geändert werden können. Dies ist für das Nachstellen von Fehlersituationen sehr wichtig. Wie Sie hierbei vorgehen, erfahren Sie in Abschnitt 4.3.3, „Änderungen vornehmen“. Im Gegensatz zur Sicht *Expressions* kann Variables aber keine komplexen Ausdrücke visualisieren.

Die Sicht „Breakpoints“

Die Sicht *Breakpoints*, die Sie in Bild 4.11 sehen, fasst alle Breakpoints der Projekte des aktuellen Arbeitsbereichs zusammen. Vereinfacht ausgedrückt, sind Breakpoints Positionen innerhalb des Quelltexts, an denen der Debugger den Programmablauf stoppt oder stoppen kann. Eclipse kennt zahlreiche Arten von Breakpoints oder Haltepunkten – ich werde sie Ihnen ausführlich in Abschnitt 4.2.2, „Breakpoints“, vorstellen.

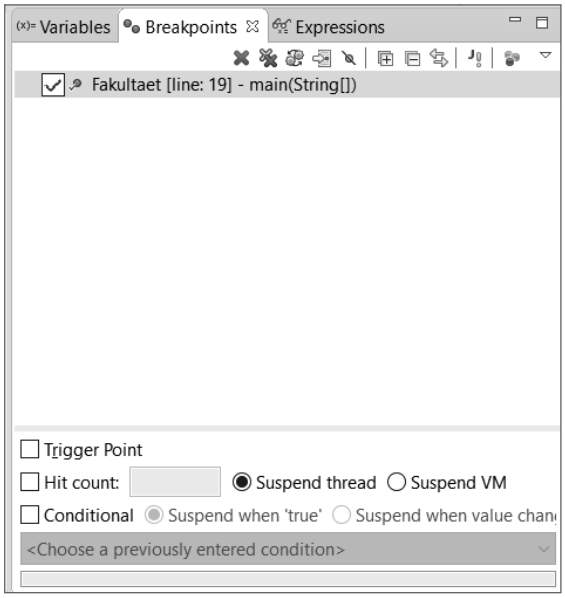

Bild 4.11 Die Sicht „Breakpoints“

Da die Sicht *Breakpoints* arbeitsbereichsbezogen arbeitet, greift sie das Ihnen bereits bekannte Konzept der Working Sets auf. Sie können durch Auswahl eines Working Sets also beispielsweise festlegen, welche Breakpoints die Sicht anzeigt.

4.2 Konzepte und Vorgehensweisen

Im vorigen Abschnitt haben Sie die Sichten der Perspektive *Debug* kennengelernt. Um sie effizient zu nutzen, ist ein fundiertes Verständnis der Konzepte und Techniken zur Fehlersuche nötig. Die Grundlagen hierfür möchte ich Ihnen im Folgenden vermitteln.

4.2.1 Architektur des Eclipse Debuggers

Das JDK beinhaltet einen remotefähigen Debugger. Dies bedeutet, dass das zu analysierende Programm auf einer anderen Maschine als auf Ihrem Entwickler-PC ausgeführt werden kann.

Client-Server-Design

Um Remote-Fähigkeit zu ermöglichen, wurde der Debugger als Client-Server-System konzipiert. Der Debug-Client läuft innerhalb der Workbench, also der Eclipse-Instanz auf Ihrer lokalen Entwickler-Maschine. Der Serverteil des Debuggers hingegen wird auf demjenigen

Rechner gestartet, auf dem auch die Anwendung, die Sie untersuchen möchten, ablaufen wird. Dies kann ebenfalls Ihr lokaler PC sein, aber eben auch ein entfernter, über das Netzwerk erreichbarer Rechner. Man unterscheidet also zwischen lokaler und entfernter Fehlersuche. Erstere ist die einfachste und gleichzeitig häufigste Form des Debuggens. Sie haben sie in Abschnitt 4.1.1, „Ein erstes Beispiel", bereits ausprobiert: Nachdem Sie die Programmierarbeiten abgeschlossen haben, starten Sie den Debug-Vorgang, indem Sie einen der entsprechenden Befehle im Menü **Run** aufrufen. Eclipse stellt automatisch eine Verbindung zwischen dem Debug-Client und Ihrer Anwendung her, die Sie nun durch das Setzen von Breakpoints, Untersuchen von Variablen und Auswerten von Ausdrücken analysieren können.

Remote Debugging bedeutet also, nach Fehlern in entfernt ablaufenden Programmen zu suchen. Dies ist natürlich besonders für klassische Enterprise-Anwendungen interessant, die auf spezielle Applikationsserver verteilt werden. Diese Art der Fehlersuche wird aber auch eingesetzt, wenn auf den Zielsystemen kein Eclipse installiert werden kann oder keine Ein-/Ausgabegeräte angeschlossen sind. Um eine Anwendung remote debuggen zu können, müssen Sie sie in einem speziellen Modus starten, damit sie auf eine Verbindung mit dem Debug-Client wartet. Außerdem muss die entfernte virtuelle Maschine diesen Mechanismus unterstützen.

Vorbereitungen für eine Remote-Debugging-Sitzung

Um Fehler in einer entfernten Anwendung zu suchen, sind grundsätzlich die folgenden Schritte nötig:

1. Stellen Sie sicher, dass die erzeugten Klassendateien Debug-Informationen enthalten. Die entsprechenden Einstellungen nehmen Sie auf der Seite **Java/Compiler** des Dialogs *Preferences* im Abschnitt Classfile Generation vor.
2. Übertragen Sie Ihre fertige Anwendung auf das Zielsystem.
3. Legen Sie eine Remote Launch Configuration an.
4. Starten Sie das Programm im Debug-Modus und geben Sie die Port-Nummer aus der Launch Configuration für die Kommunikation mit dem Client an.
5. Starten Sie den Debug-Vorgang.

Sie legen eine Remote Launch Configuration an, indem Sie mit **Run/Debug Configurations** den Ihnen bereits bekannten Dialog zum Anlegen und Bearbeiten von Konfigurationen öffnen. Er ist in Bild 4.12 zu sehen. Klicken Sie mit der rechten Maustaste auf **Remote Java Application** und wählen Sie **New Configuration**. Geben Sie der Konfiguration einen Namen, beispielsweise „Debug-Test", und setzen Sie den Connection Type auf **Standard (Socket Attach)**. Unter Connection Properties tragen Sie den Namen oder die IP-Adresse derjenigen Maschine ein, auf der die zu debuggende Anwendung ausgeführt wird. Wenn Sie das Programm auf Ihrem Entwicklungsrechner debuggen möchten, verwenden Sie localhost. Als Port tragen Sie „1044" ein. Klicken Sie nun auf **Apply** und schließen Sie den Dialog mit **Close**.

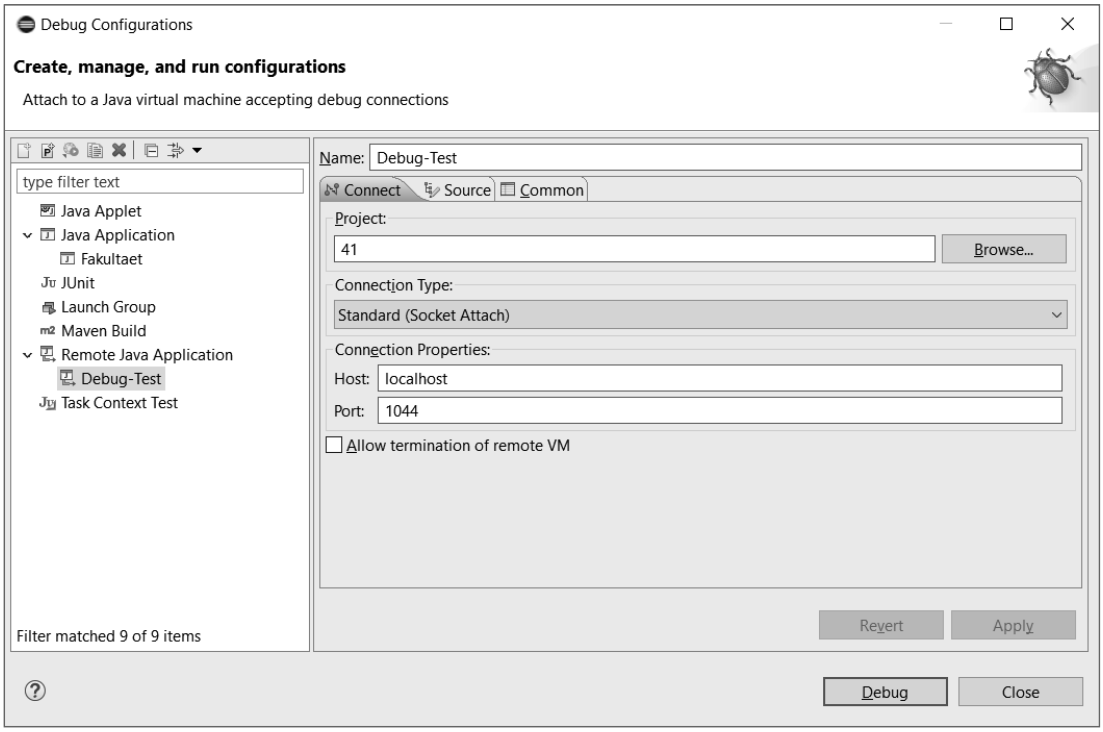

Bild 4.12 Anlegen einer neuen Remote Launch Configuration

Auch wenn Ihnen kein zweiter physischer Rechner zur Verfügung steht, können Sie das Remote Debugging ausprobieren. Wie dies funktioniert, erfahren Sie im folgenden Abschnitt.

Ein Beispiel für Remote Debugging

Ich möchte Ihnen das Remote Debugging am Beispiel der Klasse `Quadrat` demonstrieren. Sie liest eine Zahl ein und gibt deren Quadrat aus. Dies geschieht so lange, bis Sie anstelle einer Zahl „q“ eingeben.

Listing 4.2 Die Klasse „Quadrat“

```
package com.thomaskuenneth.eclipse_book;

import java.io.Console;

public class Quadrat {
  public static void main(String[] args) {
    Console c = System.console();
    if (c != null) {
      while (true) {
        c.printf("Bitte geben Sie eine Zahl ein. q beendet das Programm: ");
        String eingabe = c.readLine();
        if (eingabe.equalsIgnoreCase("q")) {
          break;
        }
```

```
            int zahl = Integer.parseInt(eingabe);
            c.printf("Das Quadrat von %d ist %d\n", zahl, zahl * zahl);
          }
        }
      }
    }
```

Wenn Sie das Programm in Eclipse starten, beendet es sich wieder, ohne eine Eingabe entgegenzunehmen und deren Quadrat zu berechnen. Dies liegt daran, dass innerhalb der Workbench keine kompatible Konsole zur Verfügung steht. Die entsprechende `if`-Abfrage liefert in diesem Fall `null`.

Öffnen Sie nun den Kommandozeileninterpreter Ihres Betriebssystems, also die Eingabeaufforderung oder Power Shell unter Windows bzw. die bash unter Linux und macOS. Wenn Sie Java so eingerichtet haben, wie ich es in Kapitel 1, „Hands-on Eclipse“, beschrieben habe, können Sie das Programm ausführen, indem Sie zunächst in das Verzeichnis bin unterhalb des Projektverzeichnisses wechseln und anschließend „java com.thomaskuenneth.eclipse_book.Quadrat“ eingeben. Ist der normale Programmstart erfolgreich verlaufen, können Sie nun den Debug-Modus starten. Geben Sie hierzu die folgende Anweisung in einer Zeile ein:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=1044
com.thomaskuenneth.eclipse_book.Quadrat
```

Sie startet das Programm und setzt die zu Ihrer Debug-Konfiguration passenden Einstellungen, nämlich Port 1044 und eine Verbindung über Sockets. Diese werden, wie Sie in Bild 4.13 sehen, nochmals ausgegeben.

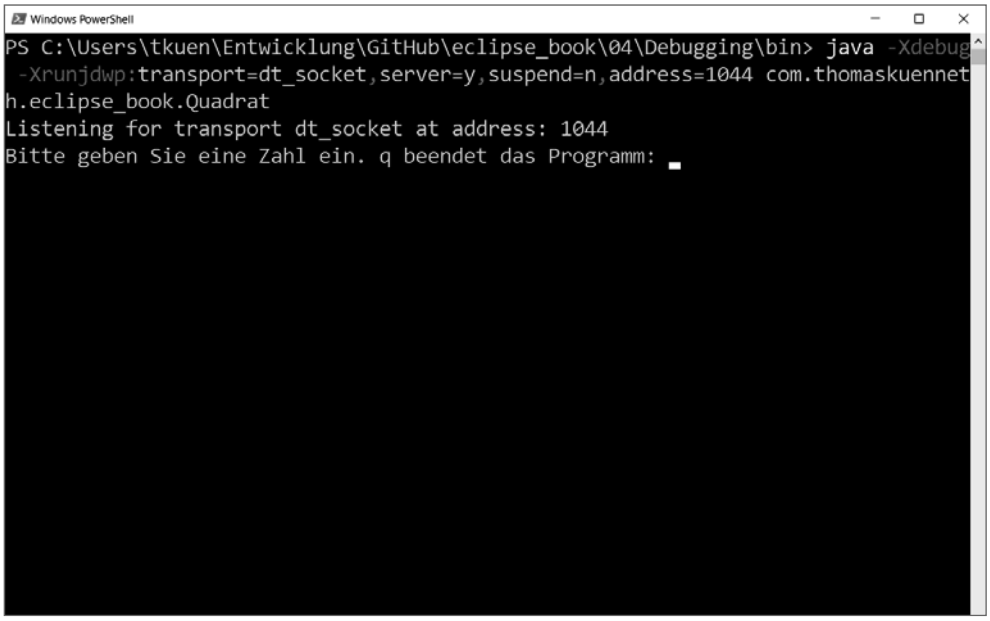

Bild 4.13 Start des Debug-Vorgangs in der Shell

Wechseln Sie zu Eclipse und legen Sie einen Breakpoint an. Bewegen Sie hierzu den Mauszeiger auf Höhe der Zeile `if (eingabe.equalsIgnoreCase("q")) {` in den linken Randbereich des Java-Editors. Nach einem Doppelklick erscheint dort ein kleines Symbol. Der Tooltip identifiziert es als Line breakpoint. Starten Sie nun den Debug-Vorgang, indem Sie **Run/Debug Configurations** aufrufen. Nach einem Doppelklick auf die von Ihnen angelegte Remote Debug Configuration wechselt die IDE in die Perspektive *Debug*.

Sie können den Zustand des Programms Quadrat in der Sicht *Debug* beobachten. Sofern Sie die Anzeige von System Threads über das Klappmenü der Sicht nicht eingeschaltet haben, sehen Sie nur einen laufenden Thread. Sie können ihn anhalten, indem Sie auf das Toolbar-Symbol (**Suspend**) klicken oder per Rechtsklick auf den Thread ein Kontextmenü öffnen und den gleichnamigen Befehl auswählen.

TIPP: Sie können einen Breakpoint auch anlegen, indem Sie im linken Randbereich des Java-Editors mit der rechten Maustaste in dieje

NIGE Zeile klicken, die den Breakpoint enthalten soll. Wählen Sie dann **Toggle Breakpoint**.

Wie eine Zeile nach Anlegen eines Haltepunkts aussieht, sehen Sie in Bild 4.14. In diesem Beispiel wird die Programmausführung nach dem Einlesen der Eingabe unterbrochen.

Bild 4.14 Java-Editor mit einem gesetzten Breakpoint

Da Sie den Breakpoint nach dem Start des Programms angelegt haben, wartet die Klasse schon auf eine Eingabe. Wechseln Sie also in das Konsolenfenster und geben Sie eine Zahl ein. Nachdem Sie **Enter** gedrückt haben, ändert sich der in der Sicht *Debug* angezeigte Status auf Suspended. Um die Eingabe auszuwerten, rufen Sie den Befehl **Step Over** auf.

Eclipse hält nach der Prüfung auf „q“ erneut an. Sie können nun in die Sicht *Variables* wechseln und sich dort den Inhalt der Variablen `eingabe` ansehen.

Sie beenden den Debug-Vorgang, indem Sie in der Sicht *Debug* auf das Symbol (Disconnect) klicken. Falls Sie über einen zweiten Rechner verfügen, können Sie mit diesem das Remote Debugging erneut üben. Kopieren Sie die Datei Quadrat.class auf diesen Rechner und tragen Sie in der Remote Debug Configuration anstelle von localhost seine IP-Adresse ein. Um diese zu ermitteln, geben Windows-Nutzer in der Eingabeaufforderung ipconfig ein. Das Äquivalent unter Linux und macOS lautet ifconfig. Alle übrigen Schritte sind identisch.

Das in diesem Abschnitt beschriebene Remote Debugging ist auf das Senden und Empfangen von Daten über das Netzwerk angewiesen. Wenn der Client keine Verbindung mit dem Debug-Server aufbauen kann, sollten Sie prüfen, ob die von Ihnen eingesetzte Firewall unter Umständen den Datenverkehr blockiert. In diesem Fall müssen Sie den Port 1044 öffnen.

4.2.2 Breakpoints

Sie haben im vorherigen Abschnitt Haltepunkte (engl. Breakpoints) als Stellen im Quelltext kennengelernt, an denen der Debugger die Programmausführung unterbricht. Eclipse kennzeichnet aktive Breakpoints im linken Randbereich des Java-Editors mit einem blauen, inaktive mit einem weißen Kreis. Außerdem werden Breakpoints in der gleichnamigen Sicht angezeigt.

Line und Method Breakpoints

Breakpoints können auf verschiedene Weise angelegt werden. Sehr bequem ist es, im linken Randbereich der gewünschten Zeile einen Doppelklick auszulösen. War an dieser Stelle schon ein Breakpoint vorhanden, wird er gelöscht. Dasselbe Resultat erreichen Sie, indem Sie die Kontextmenüfunktion **Toggle Breakpoint** (eventuell erneut) aufrufen. Möchten Sie verhindern, dass Eclipse an einem Breakpoint anhält, ohne ihn deswegen gleich zu löschen, wählen Sie stattdessen **Disable Breakpoint**. **Enable Breakpoint** aktiviert ihn wieder. Auch das Menü Run enthält Befehle, mit denen Sie Breakpoints anlegen und löschen können.

Neben den aus dem Kontextmenü des Java-Editors bekannten Befehlen finden Sie hier zusätzlich **Toggle Line Breakpoint** und **Toggle Method Breakpoint**. Erstere haben Sie bereits kennengelernt. Sie können in Zeilen mit ausführbaren Java-Anweisungen gesetzt werden. Die Programmabarbeitung wird vor dem Erreichen der betreffenden Zeile gestoppt. Method Breakpoints hingegen beziehen sich auf das Betreten oder Verlassen von Methoden. Um das Verhalten dieses Breakpoint-Typs auszuprobieren, entfernen Sie zuerst alle Breakpoints der Klasse `Fakultaet`. Wählen Sie hierzu **Run/Remove All Breakpoints**. Bestätigen Sie die Rückfrage mit **Yes**. Markieren Sie anschließend in der Sicht *Outline* die Methode `fakultaet()` und klicken Sie dann auf **Run/Toggle Method Breakpoint**. Wie Sie in Bild 4.15 sehen, erscheint im Randbereich das Symbol eines aktivierten Breakpoints.

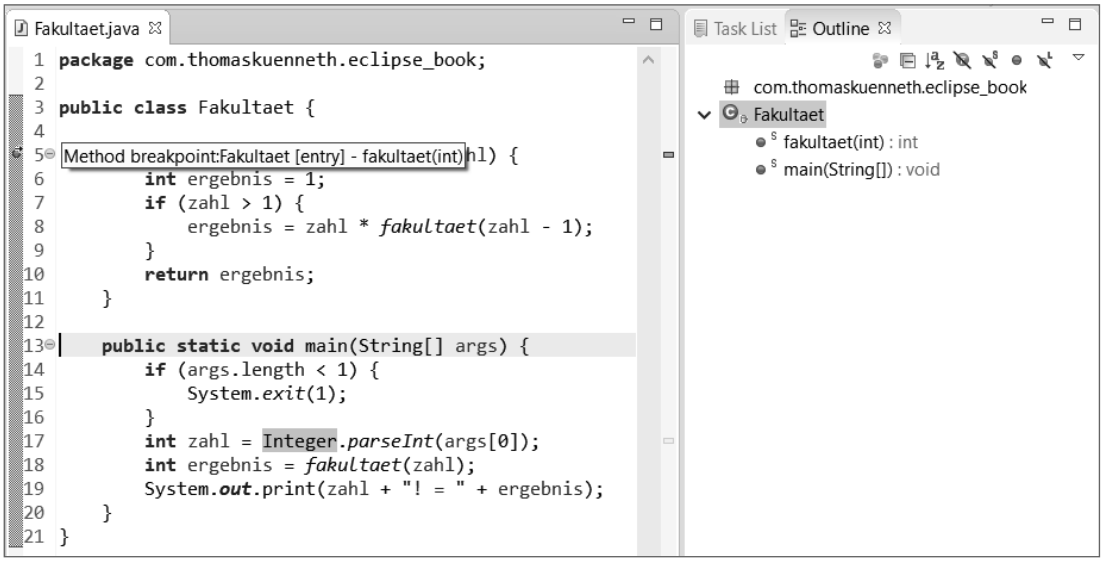

Bild 4.15 Methode mit aktivem Method Breakpoint

Starten Sie nun den Debug-Vorgang. Wenn Sie die Debug-Konfiguration aus dem ersten Abschnitt verwenden, hält Eclipse in der `main()`-Methode an, also vor dem Erreichen des neuen Breakpoints. Setzen Sie in diesem Fall die Ausführung mit **Resume** fort. Der Method Breakpoint sorgt dafür, dass das Programm vor dem Ausführen der ersten Zeile der Methode `fakultaet()` erneut gestoppt wird. Wechseln Sie nun in die Sicht *Breakpoints* und wählen Sie im Kontextmenü des Method Breakpoints **Breakpoint Properties**. Sie sehen den in Bild 4.16 gezeigten Eigenschaften-Dialog, über den Sie das Verhalten des Breakpoints beeinflussen können. Was es mit **Hit count** und **Conditional** auf sich hat, verrate ich Ihnen in Abschnitt 4.3.1, „Bedingte Programmunterbrechungen“.

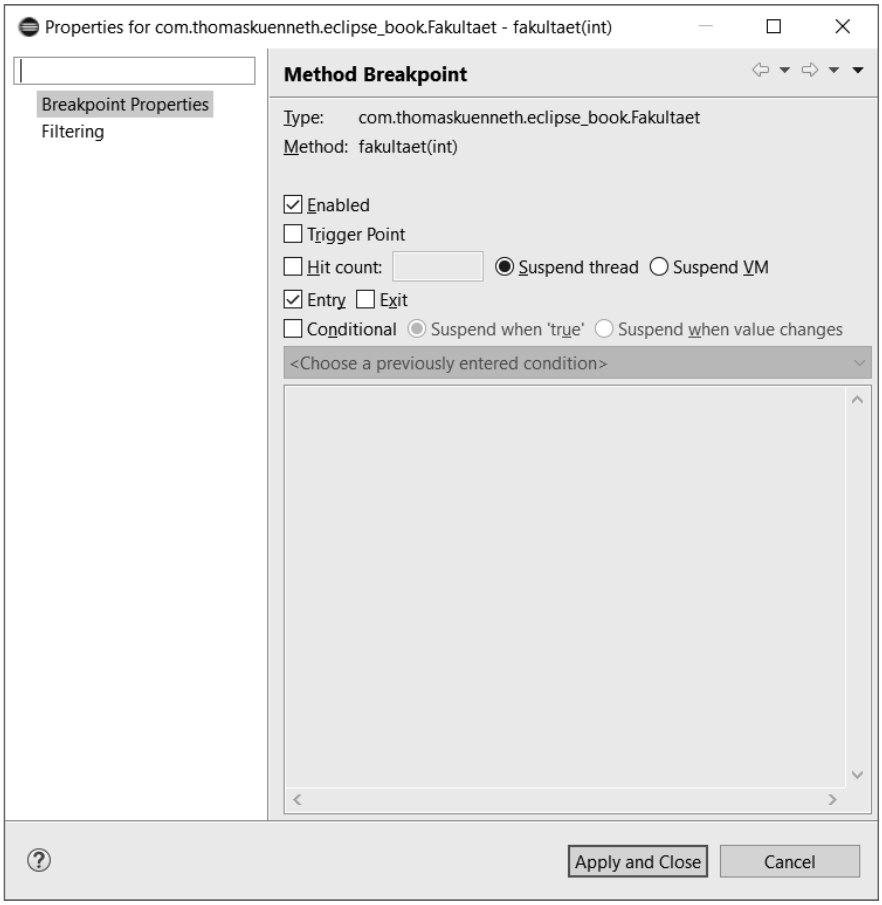

Bild 4.16 Eigenschaften eines Method Breakpoints

Im Augenblick soll es nur um die Checkboxen **Entry** und **Exit** gehen. Hier geben Sie an, wann der Programmablauf angehalten werden soll. Die Verhaltensweise bei **Entry** haben Sie soeben gesehen: Eclipse stoppt vor dem Abarbeiten der ersten Programmzeile der Methode. **Exit** hingegen wirkt nach der letzten ausgeführten Zeile, also unmittelbar vor der Rückkehr in den aufrufenden Programmteil. Um auch dieses Verhalten auszuprobieren, setzen Sie ein Häkchen vor **Exit**, entfernen das vor **Entry**, schließen den Dialog mit **Apply and Close** und führen das Programm mit **Resume** weiter aus. Die Positionen, an denen Eclipse die Ausführung des Programms anhalten soll, lassen sich übrigens auch direkt über die Befehle **Exit** und **Entry** des Kontextmenüs eines Method Breakpoints in der Sicht *Breakpoints* festlegen.

Die beiden bisher vorgestellten Breakpoint-Typen wirken, vereinfacht ausgedrückt, auf Zeilen oder Bereiche eines Programms. Sogenannte Watchpoints beziehen sich hingegen auf Variablen. Genauer gesagt, können deren Statusänderungen die Programmausführung unterbrechen.

Watchpoints

Ich möchte Ihnen die Funktionsweise von Watchpoints anhand der Klasse `WatchPointTest` vorstellen. Sie können das kurze Programm entweder wie folgt eingeben oder aus den Materialien zum Buch (*https://github.com/tkuenneth/eclipse_book*) kopieren.

Listing 4.3 WatchPointTest.java

```
package com.thomaskuenneth.eclipse_book;

public class WatchPointTest {
  private static int test1;
  private int test2;

  public WatchPointTest() {
    test1 = 1;
    test2 = test1;
    System.out.println(test2);
  }

  public static void main(String[] args) {
    new WatchPointTest();
  }
}
```

Um einen Watchpoint anzulegen, setzen Sie den Cursor innerhalb einer Variablendeklaration auf den Namen der betreffenden Variable und rufen anschließend **Run/Toggle Watchpoint** auf. Noch schneller geht es über die Sicht *Outline*. Sie erreichen sie über **Window/Show View**. Klicken Sie die Variable einfach mit der rechten Maustaste an und wählen dann **Toggle Watchpoint**.

Legen Sie auf diese Weise bitte einen Watchpoint für `test1` an. Die Zeile wird, wie Bild 4.17 zeigt, im linken Randbereich mit einem entsprechenden Symbol versehen. Starten Sie nun den Debug-Vorgang. Sie müssen hierzu übrigens keine eigene Debug-Konfiguration anlegen. Öffnen Sie einfach das Kontextmenü des Java-Editors und wählen Sie **Debug As/Java Application**.

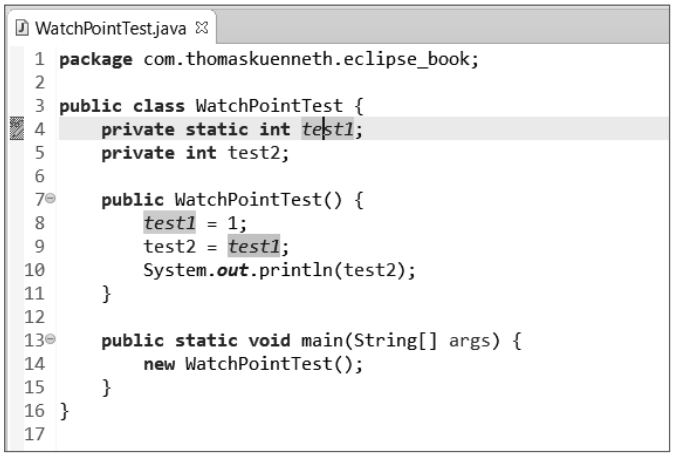

Bild 4.17 Watchpoint auf der Variablen „test1“

Die Programmausführung wird vor der Zuweisung des Werts „1“ an `test1` angehalten. Lassen Sie das Programm mit **Resume** weiterlaufen. Sie werden feststellen, dass es schon in der nächsten Zeile erneut gestoppt wird. Dies mag auf den ersten Blick verwirrend sein, denn auf die Variable `test2` haben Sie keinen Watchpoint gesetzt. Die Erklärung liegt in den Eigenschaften des Watchpoints für `test1`. Fahren Sie in den Randbereich der Zeile, die die Deklaration der Variablen `test1` enthält. Wie Sie dem Text des nun erscheinenden Tooltips entnehmen können, greifen Watchpoints standardmäßig sowohl bei lesenden als auch bei schreibenden Zugriffen (access and modification). Das Programm wurde also ein zweites Mal gestoppt, weil für die Zuweisung an die nicht beobachtete Variable `test2` der aktuelle Wert von `test1` ermittelt werden musste.

Öffnen Sie bitte das Kontextmenü des Watchpoints und wählen Sie **Breakpoint Properties**. Entfernen Sie in dem Eigenschaften-Dialog das Häkchen vor **Access** (Bild 4.18) und schließen Sie den Dialog mit **Apply and Close**. Um die geänderten Einstellungen zu testen, wechseln Sie in die Sicht *Debug* und wählen im Kontextmenü **Terminate and Relaunch**.

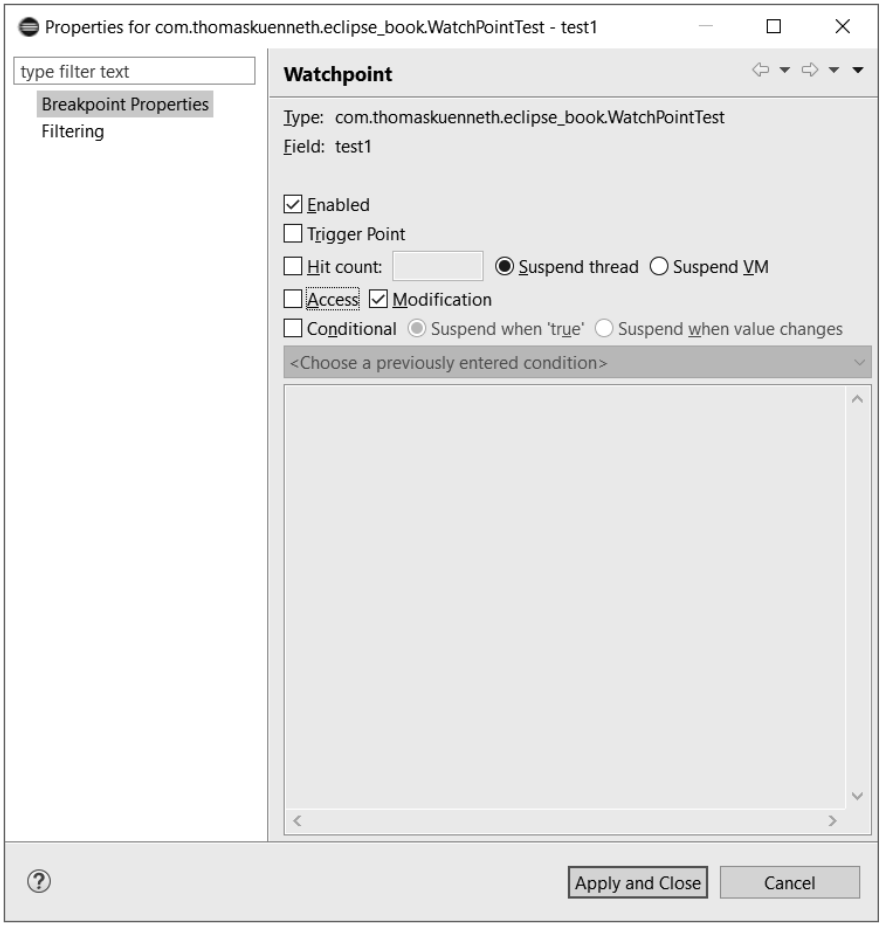

Bild 4.18 Eigenschaften von Watchpoints

Wie beim ersten Versuch hält das Programm vor der Zuweisung des Werts „1“ an. Nach einem **Resume** wird es aber ohne weitere Unterbrechung ausgeführt. Watchpoints sind also ideal, um festzustellen, welche Programmteile auf Variablen zugreifen. Beachten Sie aber, dass Sie sie nur bei Klassen- und Instanzvariablen einsetzen können, nicht aber bei lokalen Variablen.

Im folgenden Abschnitt stelle ich Ihnen einen auf den ersten Blick ungewöhnlichen, aber sehr interessanten Breakpoint-Typ vor. Indem Sie Exception Breakpoints setzen, können Sie nämlich die Programmausführung beim Auftreten von Ausnahmen unterbrechen.

Exception Breakpoints

Viele Programme enthalten in `catch{}`-Blöcken zu Debug-Zwecken Aufrufe der Methode `printStackTrace()`. Wie dies aussehen kann, zeigt das folgende Listing:

Listing 4.4 ExceptionBreakpointTest.java

```
package com.thomaskuenneth.eclipse_book;

public class ExceptionBreakpointTest {
  public static void main(String[] args) {
    try {
      for (int i = 10; i >= 0; i--) {
        System.out.println(i + " / " + i + " = " + i / i);
      }
    } catch (Throwable thr) {
      thr.printStackTrace();
    }
  }
}
```

Wenn Sie das Programm starten, werden Sie feststellen, dass es nach der Ausgabe einiger Berechnungen eine `ArithmeticException` wirft. Die Methode `printStackTrace()` nennt zusätzlich den Grund der Ausnahme (eine Division durch 0) sowie die Nummer der Zeile, in der diese aufgetreten ist. Um einen Blick auf die beteiligten Variablen werfen zu können, scheint es nahezuliegen, in der Zeile `thr.printStackTrace()` einen Line Breakpoint zu setzen. Allerdings ist diese Vorgehensweise nur bei zu erwartenden Ausnahmen sinnvoll einsetzbar. Klassische Beispiele hierfür sind die Dateioperationen der Klassenbibliothek, die `IOExceptions` werfen. Programmteile wie Berechnungen, in denen Sie keine Ausnahmen erwarten, werden Sie hingegen nicht mit `try {} catch {}`-Blöcken klammern, zumal die richtige Vorgehensweise hier das Prüfen der Formelparameter wäre. Wie aber können Sie dann den Grund der Ausnahme herausfinden?

Exception Breakpoints unterbrechen die Programmausführung, sobald eine überwachte Ausnahme geworfen wurde. Um einen solchen Haltepunkt anzulegen, wählen Sie **Run/Add Java Exception Breakpoint**. In dem in Bild 4.19 gezeigten Dialog *Add Java Exception Breakpoint* wählen Sie diejenige Ausnahme aus, die Sie überwachen möchten. Mit der Filterzeile am oberen Rand können Sie die Liste der angezeigten Ausnahmen einschränken. Suchen Sie nach `ArithmeticException` und klicken Sie diese dann an. Bevor Sie den Dialog mit **OK** schließen, prüfen Sie, ob vor den beiden Schaltern **Suspend on caught exceptions** und **Suspend on uncaught exceptions** jeweils ein Häkchen gesetzt ist.

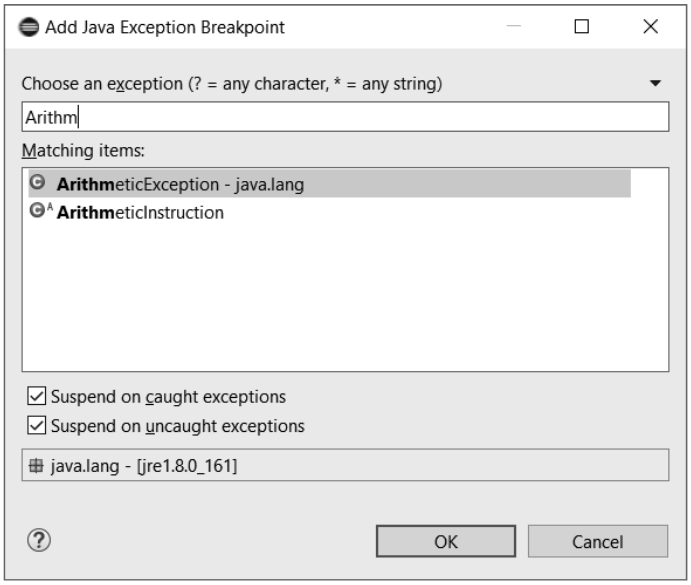

Bild 4.19 Der Dialog „Add Java Exception Breakpoint“

Starten Sie nun den Debug-Vorgang. Klicken Sie hierzu am besten mit der rechten Maustaste an eine beliebige Stelle innerhalb des Java-Editors, und wählen Sie dann **Debug As/ Java Application**. Eclipse hält das Programm in der Zeile `System.out.println(i + " / " + i + " = " + i / i);` an. Um den Grund für die Ausnahme herauszufinden, sehen Sie sich den Inhalt der beteiligten Variablen an. Bewegen Sie den Mauszeiger einfach über `i`, woraufhin sich ein kleiner Tooltip öffnet. Alternativ steht Ihnen der bereits bekannte Kontextmenübefehl **Inspect** zur Verfügung.

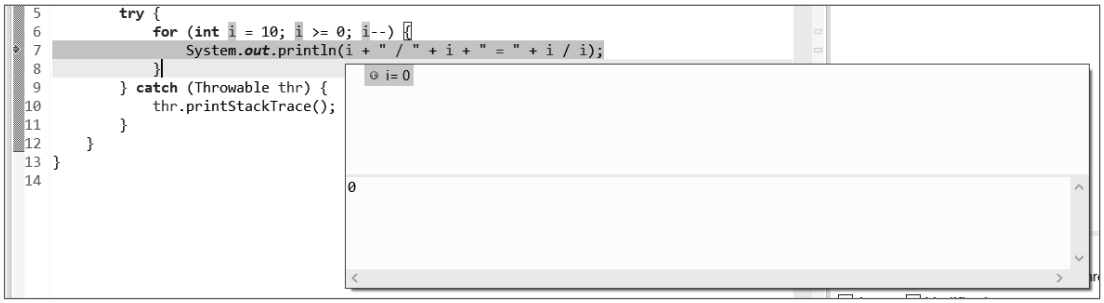

Bild 4.20 Inhalt einer Variablen anzeigen

Wie Sie in Bild 4.20 sehen, ist der Wert von `i` „0“ (zugegeben nicht besonders überraschend), was zu der bekannten Division durch 0 führt.

Auch Exception Breakpoints werden in der Sicht *Breakpoints* angezeigt. Ein Klick mit der rechten Maustaste öffnet ein Kontextmenü, das neben dem bereits bekannten Befehl **Breakpoint Properties** unter anderem die Einträge **Caught** und **Uncaught** enthält. Mit diesen Einstellungen, die Sie auch beim Anlegen des Exception Breakpoints gesetzt haben, steuern Sie das Verhalten bei `catch {}`-Blöcken, oder anders ausgedrückt, wann Eclipse beim Auf-

treten einer Ausnahme die Programmausführung unterbricht. Das Programm `ExceptionBreakpointTest` fängt `Throwable` und damit auch `ArithmeticExceptions`. Entfernen Sie das Häkchen vor **Caught**, so hält die IDE beim Erreichen der Division durch 0 nicht mehr an.

Der Halt bei nicht gefangenen Ausnahmen bietet sich an, wenn Sie das Auftreten einer Ausnahme zunächst nicht näher eingrenzen können. In diesem Fall legen Sie einen entsprechenden Exception Breakpoint an und setzen nur ein Häkchen vor **Uncaught**. Die betreffende Stelle im Quelltext können Sie sehr schön im Stack Trace der Sicht *Debug* erkennen.

Der Schalter **Uncaught** lässt sich übrigens gut mit **Subclasses of this Exception** kombinieren. Sie finden ihn ebenfalls im Eigenschaften-Dialog von Exception Breakpoints. Wenn Sie einfach nur bei allen Ausnahmen den Programmablauf unterbrechen möchten, bietet es sich nämlich an, einen Exception Breakpoint für `Throwable` anzulegen und die beiden genannten Schalter mit einem Häkchen zu versehen. Mit dem folgenden Programm können Sie das sehr gut ausprobieren:

Listing 4.5 ExceptionBreakpointTest2.java

```
package com.thomaskuenneth.eclipse_book;

public class ExceptionBreakpointTest2 {
  private static int[] werte = { 1, 2, 3 };

  public static void main(String[] args) {
    for (int i = 0; i <= werte.length; i++) {
      System.out.println(werte[i]);
    }
  }
}
```

Nachdem Sie den Breakpoint angelegt haben, starten Sie den Debug-Vorgang. Eclipse wird das Programm in der Zeile `System.out.println(werte[i]);` anhalten. Um den Grund für die Ausnahme zu ermitteln, bewegen Sie den Mauszeiger zunächst auf die Variable `werte`. Hierbei handelt es sich um ein Feld der Länge 3. Die Variable `i` hat den Wert 3, liegt also außerhalb des gültigen Bereichs für den Zugriff auf `werte`.

Mit Exception Breakpoints können Sie also sehr gut analysieren, warum eine Ausnahme aufgetreten ist. Neben dem Stack Trace der Sicht *Debug* helfen Ihnen dabei unter anderem die Sichten *Variables*, *Display* und *Expressions*.

Class Load Breakpoints

Class Load Breakpoints greifen beim erstmaligen Zugriff auf Klassen. Wie Sie sie einsetzen können, zeige ich Ihnen anhand des folgenden Programms.

Listing 4.6 ClassLoadBreakpointTest.java

```
package com.thomaskuenneth.eclipse_book;

public class ClassLoadBreakpointTest {
  public static void main(String[] args) {
    InnereKlasse i = new InnereKlasse();
    System.out.println(i);
```

```
  }
}

final class InnereKlasse {

  static {
    int i = 42;
    System.out.println(i);
  }

  protected InnereKlasse() {
    System.out.println("InnereKlasse()");
  }
}
```

Um einen Class Load Breakpoint anzulegen, klicken Sie auf **Run/Add Class Load Breakpoint**. Sie sehen daraufhin den in Bild 4.21 gezeigten Dialog *Add Class Load Breakpoint*. Darin wählen Sie diejenige Klasse aus, deren „Ladevorgang" die Programmausführung später unterbrechen soll. Geben Sie in der Filterzeile am oberen Rand des Dialogs „Inn" ein, um die Liste der darunter angezeigten Klassen einzuschränken. Markieren Sie den Eintrag `InnereKlasse` und schließen Sie den Dialog mit **OK**.

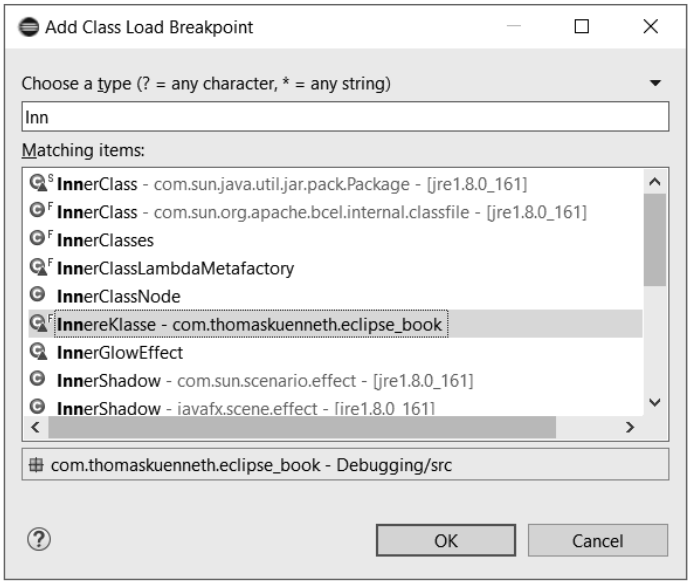

Bild 4.21 Der Dialog „Add Class Load Breakpoint"

Wenn Sie den Debug-Vorgang starten, hält Eclipse die Programmausführung in der Zeile `InnereKlasse i = new InnereKlasse();` an. Zu diesem Zeitpunkt ist die Klasse noch nicht geladen oder initialisiert worden. Mit dem Befehl **Step Into** können Sie nun alle folgenden Aktionen verfolgen, beispielsweise die Zuweisung `int i = 42;` sowie die anschließende Ausgabe. Class Load Breakpoints bieten sich also an, wenn Sie herausfinden möchten, wann eine Klasse zum ersten Mal genutzt und deshalb geladen wird.

In diesem Abschnitt haben Sie die wichtigsten Werkzeuge kennengelernt, die Eclipse für die Fehlersuche anbietet. Dabei habe ich bewusst einige Einstellmöglichkeiten und Funktionen außen vor gelassen, um Ihnen zunächst ein Gesamtbild zu vermitteln. Im nächsten Abschnitt geht es um fortgeschrittene Techniken der Fehlersuche.

4.3 Fortgeschrittene Debug-Techniken

Sehr häufig besteht die Fehlersuche aus dem schrittweisen Verfolgen des Programmflusses und dem Beobachten von Variableninhalten. Manchmal ist es aber wünschenswert, den Programmablauf nur dann zu unterbrechen, wenn eine bestimmte Situation eintritt oder eine Bedingung erfüllt ist.

4.3.1 Bedingte Programmunterbrechungen

Bisher haben Sie Breakpoints so eingesetzt, dass Eclipse den Programmablauf immer anhält, wenn der Haltepunkt erreicht wird. Sie können den Halt aber auch vom Eintreten bestimmter Bedingungen abhängig machen. Wie das geht, sehen Sie gleich.

Hit Counts

Hit Counts sorgen dafür, dass ein Programm beim Erreichen eines Breakpoints nur dann angehalten wird, wenn die Stelle im Quelltext zum wiederholten Mal passiert wurde. Hierzu ein Beispiel:

Listing 4.7 HitCountTest.java

```
package com.thomaskuenneth.eclipse_book;

public class HitCountTest {
  public static void main(String[] args) {
    for (int i = 0; i <= 2000; i++) {
      System.out.println(i);
    }
  }
}
```

Legen Sie in der Zeile `System.out.println(i);` einen Line Breakpoint an und öffnen Sie anschließend dessen Eigenschaften-Dialog. Setzen Sie nun, wie in Bild 4.22 zu sehen, ein Häkchen vor **Hit count** und tragen Sie als Wert „500“ ein. Schließen Sie den Dialog mit **Apply and Close** und starten Sie den Debug-Vorgang.

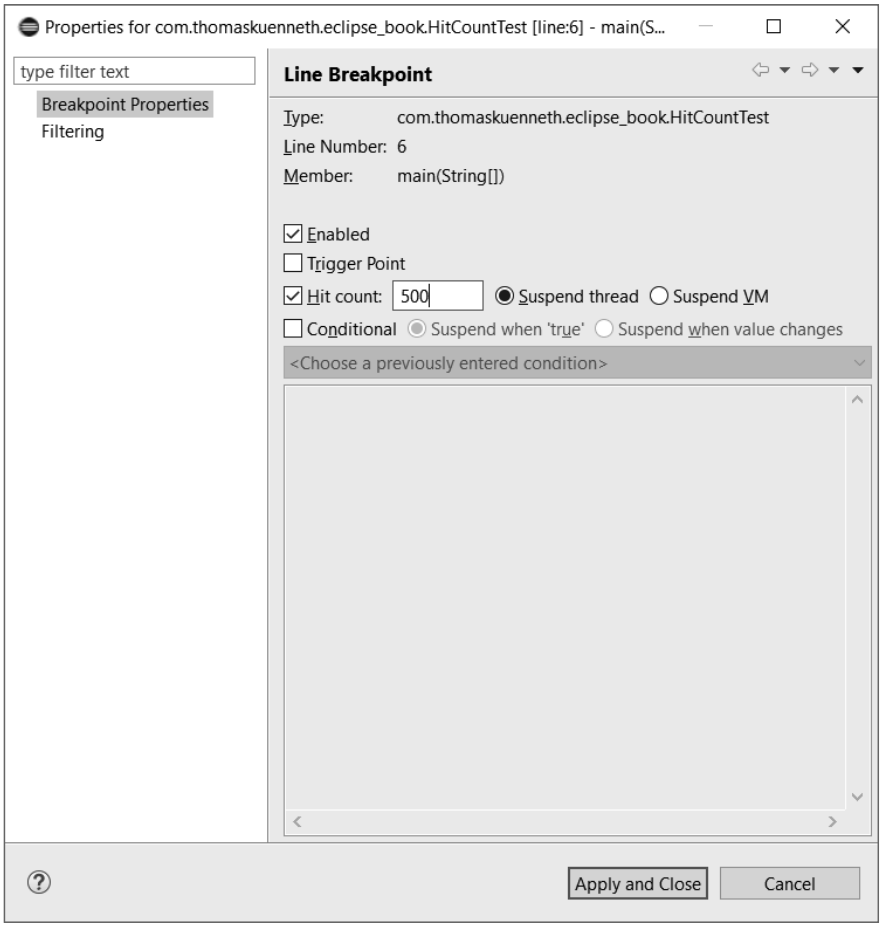

Bild 4.22 Festlegen eines Hit Counts

Eclipse hält die Programmausführung an, bevor die Zahl 499 ausgegeben wird. Zu diesem Zeitpunkt wurde der Breakpoint zum 500. Mal erreicht. Bevor Sie das Programm mit **Resume** weiter ausführen, wechseln Sie in die Sicht *Breakpoints*. Wie Sie in Bild 4.23 sehen, wurde der Breakpoint deaktiviert. Sein Häkchen ist nicht mehr gesetzt. Eclipse hält also nicht, wie Sie vielleicht erwarten würden, vor der Ausgabe von 999 an. Standardmäßig führen mit Hit Counts versehene Breakpoints nämlich höchstens einmal zu einem Programmstopp.

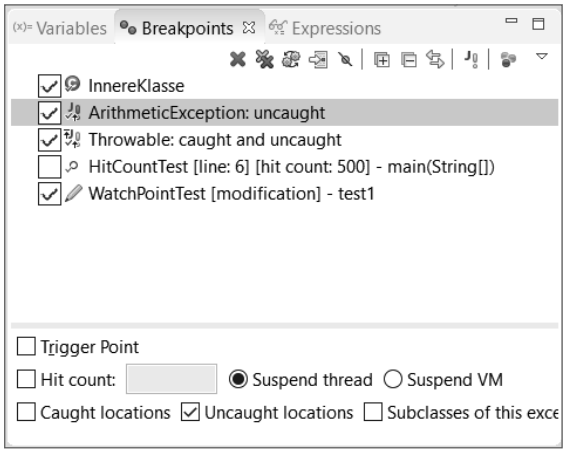

Bild 4.23 Breakpoint mit Hit Count nach dem ersten Halt

Wenn Hit Counts mehrfach wirken sollen, müssen Sie daran denken, den betreffenden Breakpoint vor dem **Resume** erneut zu aktivieren. Dies können Sie sehr einfach in der Sicht *Breakpoints* erledigen. Setzen Sie hierzu einfach wieder das Häkchen vor dem betreffenden Breakpoint oder öffnen Sie das Kontextmenü und wählen dort **Enable**.

Hit Counts sind praktisch, wenn Sie gelegentlich „nach dem Rechten" sehen möchten, wie sich also beispielsweise eine Variable innerhalb einer langen Schleife verändert. Sie lassen sich auch dann gut nutzen, wenn sich ein Fehler sehr genau vorhersagen lässt, falls Sie also schon wissen, dass er bei jedem n-ten Aufruf oder Durchlauf auftritt. Leider sind solche Konstellationen eher selten. Meistens hängen Fehlfunktionen von verschiedenen Faktoren ab, beispielsweise dem Inhalt von Instanzvariablen oder dem Ergebnis von Methodenaufrufen. Deshalb können Sie Breakpoints an Bedingungen knüpfen. Solche Bedingungen kontrollieren, wann ein Breakpoint den Programmablauf anhält. Hit Points sind, wenn Sie so wollen, eingebaute Bedingungen. Letztlich steckt hinter ihnen nämlich der Java-Ausdruck `hitPoint == zahl`, der entweder `true` oder `false` ergibt.

Bedingte Breakpoints

Auch den Einsatz von bedingten Breakpoints möchte ich Ihnen wieder an einem kleinen Programm verdeutlichen. Es ermittelt zehn Zufallszahlen und zählt, wie oft diese Zahlen kleiner oder größer als 0,5 sind.

Listing 4.8 ConditionTest.java

```
package com.thomaskuenneth.eclipse_book;

public class ConditionTest {
  public static void main(String[] args) {
    int z1 = 0;
    int z2 = 0;
    int i;
    for (i = 0; i < 10; i++) {
      double zufall = Math.random();
      if (zufall < 0.5) {
```

```
        z1++;
      } else {
        z2++;
      }
    }
    System.out.print(String.format("Von %d Zufallszahlen"
        + " sind %d < 0.5 und %d größer", i, z1, z2));
  }
}
```

Angenommen, Sie möchten überprüfen, ob das Programm richtig zählt. Außerdem würden alle Zahlen kleiner als 0,5 Sie interessieren. In diesem Fall sollte das Programm immer dann angehalten werden, wenn der Aufruf von `Math.random()` ein Ergebnis kleiner als 0,5 geliefert hat. Erzeugen Sie hierzu in der Zeile `if (zufall < 0.5) {` einen Line Breakpoint. Bevor Sie dessen Eigenschaften-Dialog öffnen, markieren Sie im Java-Editor den Text `zufall < 0.5` und legen ihn mit **Edit/Copy** auf dem Systemklemmbrett ab.

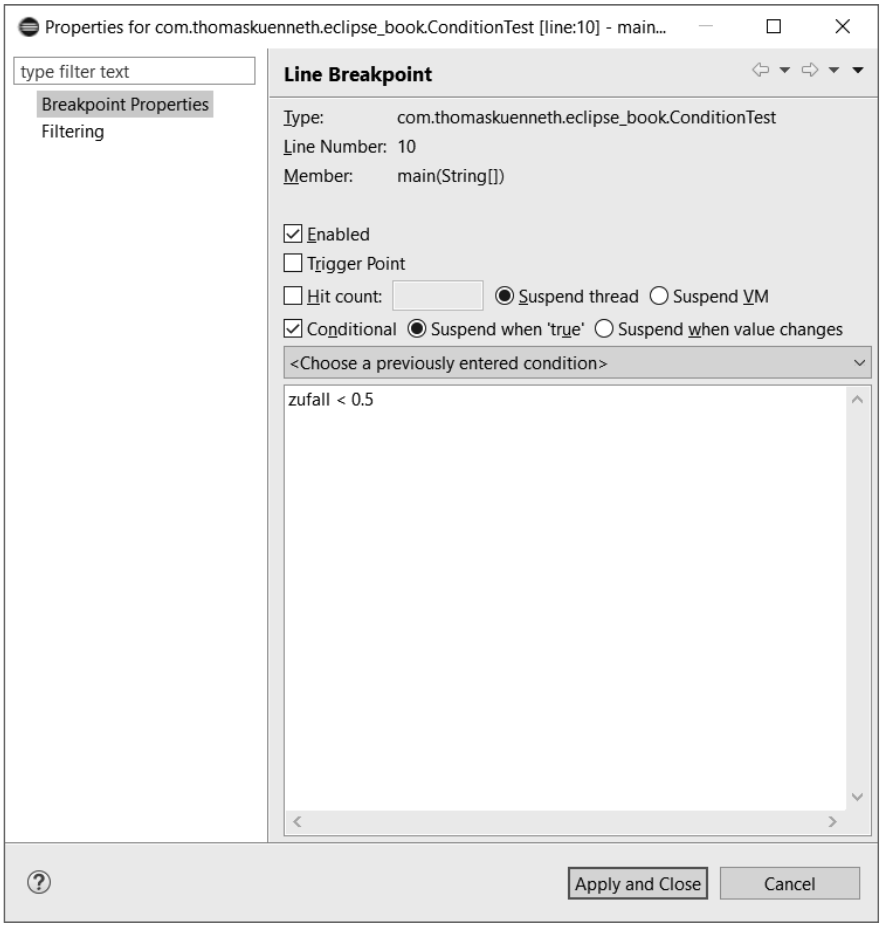

Bild 4.24 Eigenschaften eines bedingten Breakpoints

Wie Sie in Bild 4.24 sehen, müssen Sie im Eigenschaften-Dialog des Breakpoints nur ein Häkchen vor **Conditional** setzen und den auf dem Systemklemmbrett abgelegten Text in den Eingabebereich darunter einfügen. Achten Sie aber darauf, dass **Suspend when 'true'** markiert ist. Sie können nun den Debug-Vorgang starten.

Eclipse wird das Programm das erste Mal anhalten, wenn die durch die Klasse `Math` erzeugte Zufallszahl kleiner als 0,5 ist. Jetzt können Sie auf die bereits bekannte Weise den Inhalt von `zufall` und `z1` (diese Variable zählt alle Ergebnisse kleiner als 0,5) untersuchen. Mit **Resume** führen Sie `ConditionTest` weiter aus. Da `z1` beim ersten Halt mit 0 richtig vorbelegt war und nur dann um 1 hochgezählt wird, wenn die `if`-Bedingung erfüllt ist, zählt das Programm also richtig. Übrigens könnten Sie dies auch überprüfen, indem Sie zählen, wie oft Sie **Resume** aufrufen. Sie können also den Programmhalt an die Bedingung knüpfen, dass das Ergebnis eines (übrigens nahezu beliebigen) Ausdrucks `true` ergibt.

Eclipse sieht aber auch vor, Breakpoints auf der Basis von Statusänderungen auszulösen. Dies funktioniert wie folgt: Wird ein solcher bedingter Breakpoint erreicht, wertet die IDE den ihm zugewiesenen Ausdruck aus. Weicht das Ergebnis vom vorangehenden Durchlauf ab, wird der Programmablauf angehalten. Sie können eine solche Bedingung beispielsweise nutzen, um herauszufinden, wie gleichmäßig verteilt die durch `Math.random()` gelieferten Zufallszahlen sind. Wenn nämlich zwei aufeinanderfolgende Zahlen größer (oder kleiner) als 0,5 sind, wird der Ausdruck `zufall < 0.5` in beiden Fällen dasselbe Ergebnis liefern, also keine Programmunterbrechung nach sich ziehen. Indem Sie Stopps zählen, zählen Sie auch entsprechende Übergänge.

Generell sind bedingte Haltepunkte also wichtig, um unnötige Programmunterbrechungen zu vermeiden. Aus Effizienzgründen sollten Sie versuchen, nur diejenigen Programmteile in Einzelschritten zu verfolgen, die Ihnen beim Finden eines Fehlers weiterhelfen. Anders ausgedrückt: Das monotone Ablaufen der Anweisungen eines Schleifenrumpfs ist nur dann sinnvoll, falls Sie an dieser Stelle Probleme vermuten. Aber auch die Einzelschrittverarbeitung selbst lässt sich mit den Werkzeugen des Eclipse-Debuggers noch angenehmer gestalten. Wie das geht, zeige ich Ihnen im folgenden Abschnitt.

4.3.2 Kontrollierte Einzelschrittverarbeitung

Die beiden Befehle **Step Over** und **Step Into** sind wahrscheinlich die am häufigsten verwendeten Funktionen während der Fehlersuche. Viele Entwickler verwenden sie aus reiner Gewohnheit auch dann, wenn andere Techniken schneller zum Ziel führen würden.

Weitere „Step"-Befehle

Einige weitere Step-Befehle möchte ich Ihnen mit dem folgenden Programm nahebringen. Es erzeugt eine Instanz von sich selbst und gibt sowohl vorher als auch nachher die Systemzeit in Millisekunden aus, also eine fast schon triviale Angelegenheit. Es geht hier jedoch nicht darum, Fehler aufzudecken, sondern ausschließlich um das Üben der Einzelschrittkommandos.

Listing 4.9 StepTest.java

```
package com.thomaskuenneth.eclipse_book;

public class StepTest {
  public StepTest() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
  }

  public static void main(String[] args) {
    System.out.println(System.currentTimeMillis());
    new StepTest();
    System.out.println(System.currentTimeMillis());
  }
}
```

Legen Sie zunächst in der ersten Zeile der `main()`-Methode einen Line Breakpoint an und starten dann den Debug-Vorgang. Nachdem Eclipse die Ausführung unterbrochen hat, klicken Sie an eine beliebige Stelle der letzten Zeile des Methodenrumpfs (`System.out.println(System.currentTimeMillis());`), woraufhin der Text-Cursor an diese Position gesetzt wird. Rufen Sie nun **Run/Run to Line** auf. Eclipse führt das Programm weiter aus und hält in der soeben genannten Zeile wieder an. Mit **Run to Line** können Sie also ein unterbrochenes Programm fortsetzen, bis eine bestimmte Zeile erreicht wird. Das Besondere dabei ist, dass Sie hierzu nicht erst einen Breakpoint setzen müssen. Dies ist schon deshalb sehr praktisch, weil sich erfahrungsgemäß recht schnell zahlreiche Breakpoints in der Sicht *Breakpoints* ansammeln.

HINWEIS: Denken Sie daran, dass Sie mit **Run/Remove All Breakpoints** alle Haltepunkte entfernen können.

Um den nächsten Befehl auszuprobieren, müssen Sie den Debug-Vorgang neu starten. Klicken Sie hierzu im Kontextmenü der Sicht *Debug* auf **Terminate and Relaunch** und rufen Sie dann jeweils einmal **Step Over** und **Step Into** auf. Sie befinden sich nun in der ersten Zeile des Konstruktors `StepTest()`. Mit dem Befehl **Step Return**, den Sie im Menü Run sowie im Kontextmenü der Sicht *Debug* finden, können Sie die Einzelschrittverarbeitung bis zum Erreichen einer `return`-Anweisung oder des Endes einer Methode oder eines Konstruktors aussetzen. Das ist praktisch, wenn zwar die aktuelle Methode nicht weiter beobachtet werden soll, Sie nach dem Rücksprung aus ihr aber weiter in Einzelschritten arbeiten möchten.

Den umgekehrten Weg, nämlich das Aussetzen der Einzelschrittverarbeitung bis zum Erreichen eines Methoden- oder Konstruktorrumpfs, ermöglicht **Step Into Selection**. Um diese Funktion zu nutzen, markieren Sie im Java-Editor einen Methoden- oder Konstruktoraufruf (nur den Namen, nicht die sich anschließenden Klammern) und wählen dann den Eintrag **Step Into Selection** des Kontextmenüs. Damit das funktioniert, muss das Programm gestartet und anschließend unterbrochen worden sein. Sie können es auf diese Weise also nicht aufrufen und bis zu einer bestimmten Stelle ausführen.

Der letzte Befehl, den ich Ihnen hier vorstellen möchte, führt genau genommen gar keine Anweisungen aus, sondern sorgt für das kontrollierte Verlassen einer Methode. Deren Rückgabewert können Sie selbst bestimmen. Dies möchte ich Ihnen am folgenden Programm demonstrieren:

Listing 4.10 ForceReturnTest.java

```
package com.thomaskuenneth.eclipse_book;

import java.util.GregorianCalendar;

public class ForceReturnTest {
  public static void main(String[] args) {
    int i = test();
    System.out.println(i);
  }

  private static int test() {
    GregorianCalendar cal = new GregorianCalendar();
    int tag = cal.get(GregorianCalendar.DAY_OF_MONTH);
    return tag;
  }
}
```

Legen Sie einen Breakpoint in der Zeile `return tag;` an und starten Sie dann den Debug-Vorgang. Nachdem Eclipse das Programm angehalten hat, können Sie sich den Inhalt der Variablen `tag` ansehen: den aktuellen Tag des Monats. Wechseln Sie nun in die Sicht *Debug Shell* und geben Sie dort `cal.get(GregorianCalendar.MONTH) + 1` ein. Markieren Sie den eingegebenen Text und wählen Sie dann **Inspect**. Der Ausdruck liefert den aktuellen Monat als Zahl. Öffnen Sie nun das Kontextmenü und rufen Sie dort **Force Return** auf. Im Java-Editor sehen Sie, dass sich die Markierung für den nächsten auszuführenden Befehl jetzt vor der Zeile `int i = test();` befindet. Führen Sie das Programm mit **Resume** weiter aus. Ohne dass Sie an ihm Veränderungen vorgenommen haben, gibt es anstelle des Tags den aktuellen Monat aus.

Force Return ist also äußerst nützlich, falls Sie während der Fehlersuche Befehle von der Ausführung ausnehmen wollen oder müssen. Ein Grund kann sein, dass Sie eine Methode als fehlerhaft erkannt haben. In diesem Fall können Sie dem Programm den richtigen Rückgabewert „unterschieben“. **Force Return** ist aber auch praktisch, falls eine Methode auf Ressourcen zugreifen muss, die im Moment nicht zur Verfügung stehen (denken Sie an eine gestörte Netzwerkverbindung). In diesem Fall könnte die Methode den Rückgabewert nicht ermitteln. Auch hier können Sie stattdessen einen Wert vorgeben und mit der Fehlersuche fortfahren.

Wie Sie sehen, bietet Eclipse eine ganze Reihe von Befehlen für eine effiziente Einzelschrittverarbeitung. Natürlich werden Sie **Step Over** und **Step Into** am häufigsten einsetzen. Denken Sie aber daran, dass es in bestimmten Situationen unter Umständen besser geeignete Funktionen gibt. Auch der nächste Abschnitt beschäftigt sich mit der Einzelschrittverarbeitung. Ich stelle Ihnen die sogenannten Step Filter vor, mit denen Sie Klassen und Pakete „ausblenden“ können.

Step Filter

Es gibt einige Gründe, Klassen oder Pakete von der Einzelschrittverarbeitung auszuschließen. Wenn Sie beispielsweise eine Bibliothek einsetzen, deren Quelltext nicht zur Verfügung steht, ist ein **Step Into** ohnehin sinnlos. Aber selbst dann, wenn er vorhanden ist, kann es sehr verwirrend sein, plötzlich mit dem Quelltext von `java.io.PrintStream` konfrontiert zu werden, nur weil Sie in der Zeile `System.out.println(i);` versehentlich **Step Into** anstelle von **Step Over** angeklickt haben.

Um Step Filter ein- oder auszuschalten, klicken Sie in der Toolbar auf das Symbol (**Use Step Filters**) oder wählen den gleichnamigen Befehl im Kontextmenü der Sicht *Debug*. Welche Klassen und Pakete bei aktiviertem Filter von Einzelschrittbefehlen ausgenommen sind, legen Sie auf der Seite Step Filtering des Dialogs *Preferences* fest, die Sie in Bild 4.25 sehen.

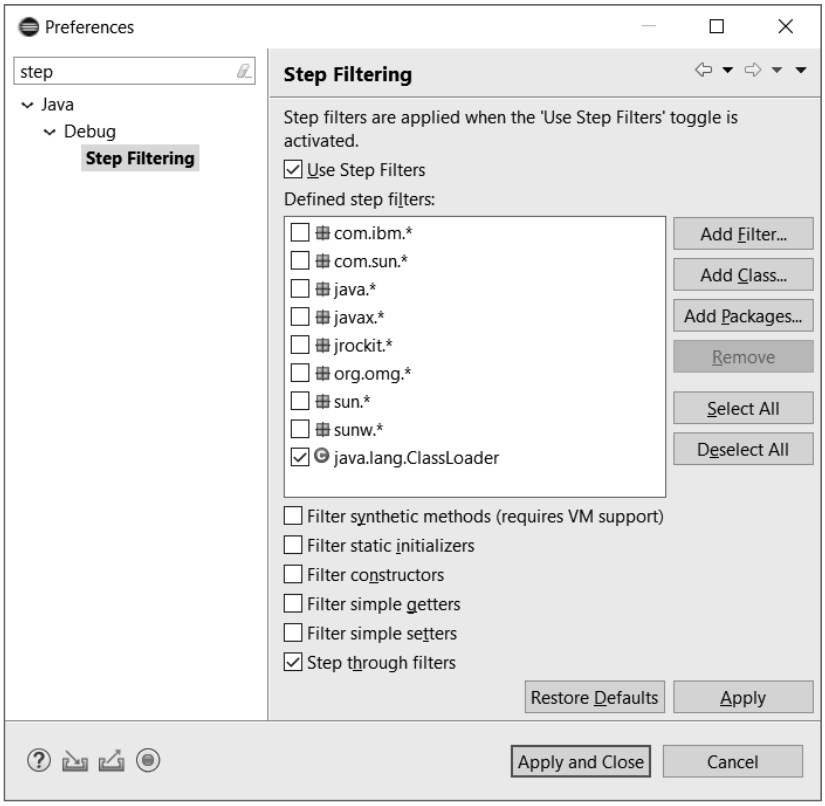

Bild 4.25 Die Seite „Step Filtering“ des Dialogs „Preferences“

TIPP: Denken Sie daran, dass Sie mit der Filterzeile des Einstellungen-Dialogs schnell zu gewünschten Seiten navigieren können.

Die Seite Step Filtering ist in mehrere Bereiche unterteilt. Die Liste der gefilterten Klassen und Pakete lässt sich nur bearbeiten, wenn ein Häkchen vor **Use Step Filters** gesetzt ist. In diesem Fall können Sie mit den entsprechenden Schaltflächen Klassen und Pakete hinzufügen oder löschen. Wie Sie dies in der Praxis einsetzen, zeige ich Ihnen gleich. Beachten Sie, dass sich Filter individuell ein- und ausschalten lassen. Selbst wenn also **Use Step Filters** aktiviert ist, greift ein Filter nur, wenn vor seinem Namen ein Häkchen gesetzt wurde. Der untere Bereich des Dialogs enthält sechs Schalter, die die Funktionsweise der Step Filter beeinflussen. Mit den ersten fünf legen Sie fest, welche Teile einer Klasse ausgeblendet werden sollen. **Step through filters** beeinflusst das Verhalten, wenn von gefilterten Klassen in Form eines Callbacks auf nicht gefilterte Bereiche zugegriffen wird.

Welche Konsequenzen dies hat, möchte ich Ihnen anhand des Programms `StepFilterTest` demonstrieren. Es erzeugt drei Objekte, die eine Zahl speichern. Da die Klasse das Interface `Comparable` implementiert, können Felder dieses Typs mit `java.util.Arrays.sort()` sortiert werden.

Listing 4.11 StepFilterTest.java

```
package com.thomaskuenneth.eclipse_book;

import java.util.Arrays;

public class StepFilterTest implements Comparable<StepFilterTest> {

  public int wert;

  public StepFilterTest(int zahl) {
    this.wert = zahl;
  }

  public static void main(String[] args) {
    // Feld erzeugen
    StepFilterTest[] feld = new StepFilterTest[3];
    feld[0] = new StepFilterTest(10);
    feld[1] = new StepFilterTest(5);
    feld[2] = new StepFilterTest(20);
    // ausgeben
    ausgabe(feld);
    // sortieren
    Arrays.sort(feld);
    // erneut ausgeben
    ausgabe(feld);
  }

  public static void ausgabe(StepFilterTest[] feld) {
    for (int i = 0; i < feld.length; i++) {
      if (i > 0) {
        System.out.print(",");
      }
      System.out.print(feld[i]);
    }
    System.out.println();
  }

  public String toString() {
```

```
    return Integer.toString(wert);
  }

  public int compareTo(StepFilterTest o) {
    if (o.wert < wert) {
      return 1;
    } else if (o.wert > wert) {
      return -1;
    }
    return 0;
  }
}
```

Setzen Sie in der verwendeten Debug-Konfiguration das Häkchen vor **Stop in main** und starten Sie anschließend den Debug-Vorgang. Nachdem Eclipse in der ersten Zeile der `main()`-Methode angehalten hat, richten Sie einen Step Filter für die Klasse `java.util.Arrays` ein. Öffnen Sie hierzu, wie weiter oben beschrieben, die Seite Step Filtering des Dialogs *Preferences* und klicken Sie auf **Add Filter**.

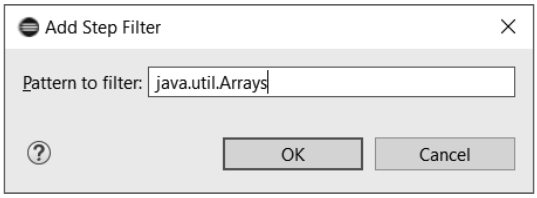

Bild 4.26 Der Dialog „Add Step Filter“

Sie sehen den in Bild 4.27 gezeigten Dialog *Add Step Filter*. Geben Sie bei Pattern to filter „java.util.Arrays“ ein und schließen Sie den Dialog mit **OK**. Die Klasse `java.util.Arrays` erscheint unter **Defined step filters** und wurde automatisch mit einem Häkchen versehen. Bevor Sie den Dialog *Preferences* mit **Apply and Close** schließen, entfernen Sie die Häkchen vor allen sechs Schaltern im unteren Bereich des Dialogs. Legen Sie nun in der Zeile `Arrays.sort(feld);` einen Breakpoint an und vergewissern Sie sich, dass **Use Step Filters** aktiviert ist.

Setzen Sie nun den Debug-Vorgang mit **Resume** fort. Nachdem Eclipse die Ausführung des Programms angehalten hat, klicken Sie auf **Step Into**. Da Arrays zur Liste der gefilterten Klassen gehört, wirkt der Befehl wie **Step Over**. Deaktivieren Sie nun **Use Step Filters** und klicken Sie im Kontextmenü der Sicht *Debug* auf **Terminate and Relaunch**. Da in der Debug-Konfiguration **Stop in main** aktiv ist, wählen Sie **Resume**. Die Ausführung stoppt wieder vor der Sortierung. Nach einem erneuten **Step Into** wird diesmal die Klasse Arrays im Java-Editor angezeigt.

Rufen Sie den Dialog *Preferences* auf, indem Sie im Kontextmenü der Sicht *Debug* die Funktion **Edit Step Filters** wählen. Schalten Sie die Filterfunktion wieder ein und aktivieren Sie alle sechs Schalter im unteren Bereich des Dialogs. Starten Sie nun mit **Terminate and Relaunch** abermals den Debug-Vorgang und klicken Sie erst auf **Resume**, dann auf **Step Into**. Die Programmausführung wird in der ersten Zeile der Methode `compareTo()` angehalten.

Step through filters ist also äußerst praktisch, wenn Ihr Programm aus gefilterten Klassen heraus aufgerufen wird. Das ist immer dann der Fall, wenn Sie Interfaces implementieren und an Klassen der Standardbibliothek übergeben müssen.

Eclipse bietet zahlreiche Möglichkeiten, die Einzelschrittverarbeitung auf diejenigen Bereiche Ihres Programms zu beschränken, die für die aktuelle Fehleranalyse tatsächlich nötig sind. Mein Rat an dieser Stelle ist, diese konsequent zu nutzen. Denn jedes unnötige **Step Into** kostet wertvolle Zeit und lenkt Ihren Blick unter Umständen von den eigentlichen Fehlerursachen ab.

4.3.3 Änderungen vornehmen

Zu Beginn dieses Kapitels habe ich erwähnt, dass Eclipse Ihnen die Möglichkeit bietet, während einer Debug-Sitzung Änderungen am gerade untersuchten Programm vorzunehmen. Vielleicht fragen Sie sich, warum Sie zu diesem Zeitpunkt überhaupt Änderungen vornehmen sollten. Gründe hierfür gibt es viele. Obwohl die Rechenleistung in den letzten Jahren dramatisch zugenommen hat, existieren immer noch viele Algorithmen, die so aufwendig sind, dass selbst moderne Computer viele Stunden oder Tage mit ihrer Abarbeitung beschäftigt sind. Während der Fehlersuche sollten solche Programme möglichst selten neu gestartet werden müssen. Oder stellen Sie sich vor, das Ergebnis einer aufwendigen Berechnung wird in anderen Programmteilen genutzt. Wenn Sie ausprobieren möchten, wie sich Ihr Programm verhält, wenn ein Aufruf ein anderes Ergebnis liefert, wäre es praktisch, diese Operation nicht jedes Mal aufs Neue durchführen zu müssen.

Variableninhalte ändern

In Abschnitt 4.3.2, „Kontrollierte Einzelschrittverarbeitung“, habe ich Ihnen gezeigt, wie Sie mithilfe der Funktion Force Return die Ergebnisse von Methodenaufrufen manipulieren können. Allerdings wird in diesem Fall überhaupt kein Befehl mehr abgearbeitet, abgesehen natürlich von der Auswertung Ihrer Ausdrücke. Wie Sie gleich sehen werden, können Sie in der Sicht *Variables* gezielt Daten verändern, die dann von Ihrem Programm weiterverarbeitet werden.

Listing 4.12 Summe.java

```
package com.thomaskuenneth.eclipse_book;

public class Summe {

  public static void main(String[] args) {
    int ergebnis = summe(args);
    System.out.println(ergebnis);
  }

  private static int summe(String[] args) {
    int wert = 0;
    for (int i = 0; i < args.length; i++) {
      int aktuell = Integer.parseInt(args[i]);
      wert += aktuell;
    }
    return wert;
  }
}
```

Die Klasse `Summe` addiert alle Argumente, die es über die Kommandozeile erhält, und gibt das Ergebnis auf der Konsole aus. Um es zu testen, könnten Sie eine eigene Debug-Konfiguration erstellen und dort einige Zahlen eintragen. Legen Sie stattdessen in der Zeile `int ergebnis = summe(args);` der `main()`-Methode einen Breakpoint an und starten Sie dann den Debug-Vorgang. Wechseln Sie nun in die Sicht *Variables* und sehen Sie sich `args` an. Da Sie beim Start keine Werte übergeben konnten, ist es ein leeres Feld ohne Elemente. Um das zu ändern, klicken Sie den Variablennamen an und wählen im Kontextmenü **Change Value**. Sie sehen den Dialog *Change Object Value*, in dem Sie den Ausdruck `(new String [] {"2", "3", "5"})` eingeben. Dies ist in Bild 4.27 zu sehen.

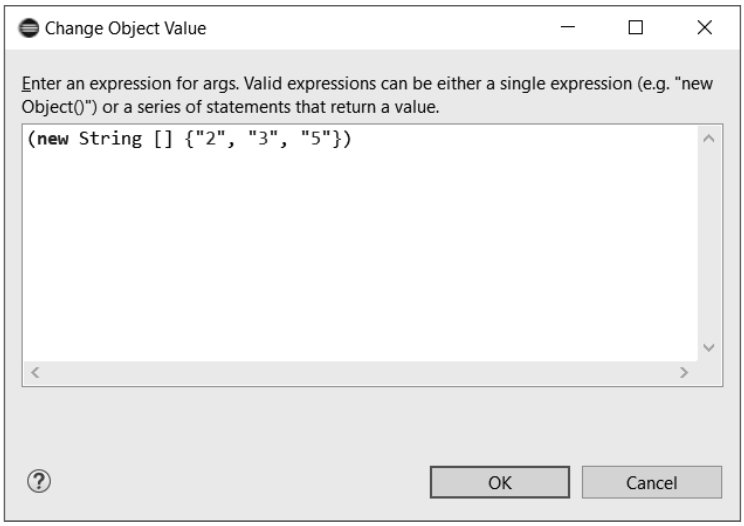

Bild 4.27 Eingeben eines Felds mit Strings

Sie haben auf diese Weise ein Feld der Länge 3 erzeugt und dessen Elemente mit Werten gefüllt. Diese können Sie in der in Bild 4.28 gezeigten Sicht *Variables* einsehen und bei Bedarf nochmals modifizieren. Die Darstellung der Sicht lässt sich über ihr Klappmenü in weiten Grenzen konfigurieren. Beispielsweise können Sie den Eingabebereich ausblenden. Dieser lässt übrigens, wie auch die Zellen der Tabelle, keine Eingabe von Java-Ausdrücken zu.

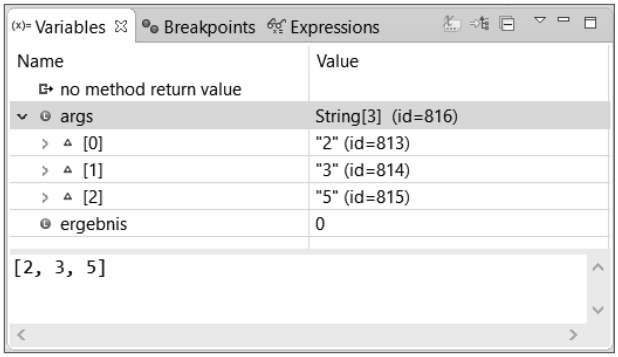

Bild 4.28 Die Sicht „Variables“ nach Ändern der Variablen „args“

Springen Sie nun mit **Step Into** in die erste Zeile der Methode `summe()`. Die Sicht *Variables* zeigt zunächst nur eine Variable an, nämlich das als Parameter übergebene Feld `args`. Nach einem **Step Over** sehen Sie zusätzlich `wert`. Wechseln Sie in die Sicht *Debug Shell* und geben dort `args = new String [] {"42"};` ein. Markieren Sie dann, wie gewohnt, den Ausdruck und rufen Sie anschließend **Execute the Selected Text** auf. Wenn Sie in die Sicht *Variables* wechseln, sehen Sie, dass Eclipse den Wert von `args` abermals geändert hat.

Durch das Ändern von Variableninhalten können Sie auf sehr bequeme Weise Datenkonstellationen simulieren und so nach Fehlern suchen oder das Programmverhalten analysieren. Allerdings mussten diese Änderungen bisher vor dem Durchlaufen des zu untersuchenden Programmfragments erfolgen. Im nächsten Abschnitt zeige ich Ihnen, wie Sie ohne Neustart des Debug-Vorgangs Codebereiche wiederholt ausführen können.

Codebereiche erneut ausführen

Das Programm Summe, das Sie im vorigen Abschnitt kennengelernt haben, ist nicht sehr fehlertolerant programmiert. Wenn sich eines der übergebenen Argumente nicht in eine Zahl umwandeln lässt, bricht das Programm mit einer Ausnahme ab. Um dies auszuprobieren, löschen Sie zunächst alle Breakpoints – außer den in der Zeile `int ergebnis = summe(args);` – und starten Sie dann den Debug-Vorgang. Nachdem Eclipse die Ausführung angehalten hat, ändern Sie die Variable `args` mit dem Ausdruck `args = new String [] {"a"}` in ein Feld mit einem Element "a". Führen Sie das Programm anschließend mit **Resume** weiter aus. Erwartungsgemäß führt "a" zu der Ausnahme `NumberFormatException`.

In Bild 4.29 sehen Sie den in der Sicht *Debug* angezeigten Stack Trace. Klicken Sie die Elemente unterhalb des Knotens Thread in der Sicht *Debug* der Reihe nach an. Sie werden bemerken, dass bestimmte Zeilen im Java-Editor hervorgehoben werden. Diese entsprechen dem eben angeklickten Element. Wenn Sie auf diese Weise die Zeile `int aktuell = Integer.parseInt(args[i]);` gefunden haben, können Sie in der Sicht *Variables* das falsche Element des Arrays korrigieren, indem Sie eine gültige Zahl eintragen. Öffnen Sie dann mit der rechten Maustaste in der Sicht *Debug* das Kontextmenü und wählen Sie **Drop To Frame**. Achten Sie darauf, dass sich die Maus über dem zuletzt angeklickten Element befindet. Wenn Sie jetzt mit **Resume** das Programm weiter ausführen, wird keine Ausnahme geworfen.

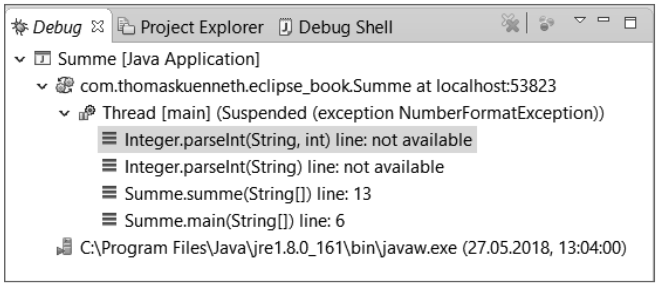

Bild 4.29 Stack Trace nach der Ausnahme „NumberFormatException“

Die Funktion **Drop To Frame** erlaubt also das „Zurückspulen“ zu einem bestimmten Punkt der Programmausführung. Hierbei werden auch die Werte lokaler Variablen in den Zustand

versetzt, den sie zum entsprechenden Zeitpunkt hatten. Die Ausführung des Programms wird am Beginn derjenigen Methode fortgesetzt, die zu dem ausgewählten Frame gehört. Sie können somit ohne Neustart des Debug-Vorgangs Programmteile mit geänderten Datenkonstellationen praktisch beliebig oft aufrufen und so das Verhalten Ihres Programms genau untersuchen.

In den bisherigen Abschnitten dieses Kapitels haben Sie gelernt, wie Sie mit dem in Eclipse eingebauten Debugger Ihre Programme zur Laufzeit untersuchen können. Im Folgenden beschreibe ich einen Ansatz, Fehler schon während der Entwicklung zu vermeiden.

4.4 Unit-Tests

Unter Unit-Tests versteht man, vereinfacht ausgedrückt, das häppchenweise Testen eines Programms. Dazu wird es in seine kleinsten, einzeln prüfbaren Einheiten zerlegt. Jede dieser Einheiten muss sogenannte Testfälle fehlerfrei durchlaufen. Trifft dies zu, kann davon ausgegangen werden, dass die getestete Einheit, wie gewünscht, funktioniert. Im Gegensatz zur Fehlersuche im Debugger wird bei Unit-Tests nicht untersucht, was während der Programmausführung passiert. Stattdessen gilt ein Test als erfolgreich absolviert, wenn die Einheit das erwartete, im Testfall spezifizierte Ergebnis liefert. Unit-Tests können und sollten übrigens sehr oft während der Entwicklung durchgeführt werden. Ein eigener Programmierstil, die sogenannte testgetriebene Entwicklung, hat sich denn auch voll und ganz dem Unit-Test verschrieben.

4.4.1 JUnit im Überblick

Unit-Tests werden in der Regel mit speziellen Test-Frameworks durchgeführt. Sie helfen, den Testvorgang zu automatisieren. Das sicher bekannteste Test-Framework für Java ist JUnit. Es wurde von Kent Beck und Erich Gamma entwickelt. Beck hatte mit SUnit bereits ein solches Framework für die Programmiersprache Smalltalk implementiert und später gemeinsam mit Gamma nach Java portiert.

Einen Testfall anlegen

JUnit ist sehr gut in Eclipse eingebunden. Anhand des unten abgedruckten Programms Schaltjahr.java möchte ich Ihnen zunächst zeigen, wie Sie innerhalb der IDE einen Testfall anlegen und ausführen.

Listing 4.13 Schaltjahr.java

```
package com.thomaskuenneth.eclipse_book;

public class Schaltjahr {
  public static void main(String[] args) {
    int jahr = Integer.parseInt(args[0]);
    boolean schaltjahr = schaltjahrPruefen(jahr);
    System.out.println(jahr + " ist "
        + (schaltjahr ? "" : "k") + "ein Schaltjahr");
  }

  public static boolean schaltjahrPruefen(int jahr) {
    boolean schaltjahr = false;
    if ((jahr % 4) == 0) {
      schaltjahr = true;
      if ((jahr % 400) != 0) {
        if ((jahr % 100) == 0) {
          // schaltjahr = false;
        }
      }
    }
    return schaltjahr;
  }
}
```

Nachdem Sie den Programmcode eingegeben oder aus den Materialien zum Buch (*https://github.com/tkuenneth/eclipse_book*) übernommen haben, können Sie das Programm auf die Ihnen bereits bekannte Weise starten und mit ein paar Jahreszahlen „spielen“. Wie Sie sicher schnell festgestellt haben, beachtet der Algorithmus nicht alle Ausnahmen richtig. Mehr dazu etwas später.

Legen Sie nun einen sogenannten Testfall an. Markieren Sie hierzu die Klasse `Schaltjahr` im Package Explorer und klicken Sie auf **File/New/JUnit Test Case**. Sie sehen daraufhin den in Bild 4.30 gezeigten Dialog *New JUnit Test Case*.

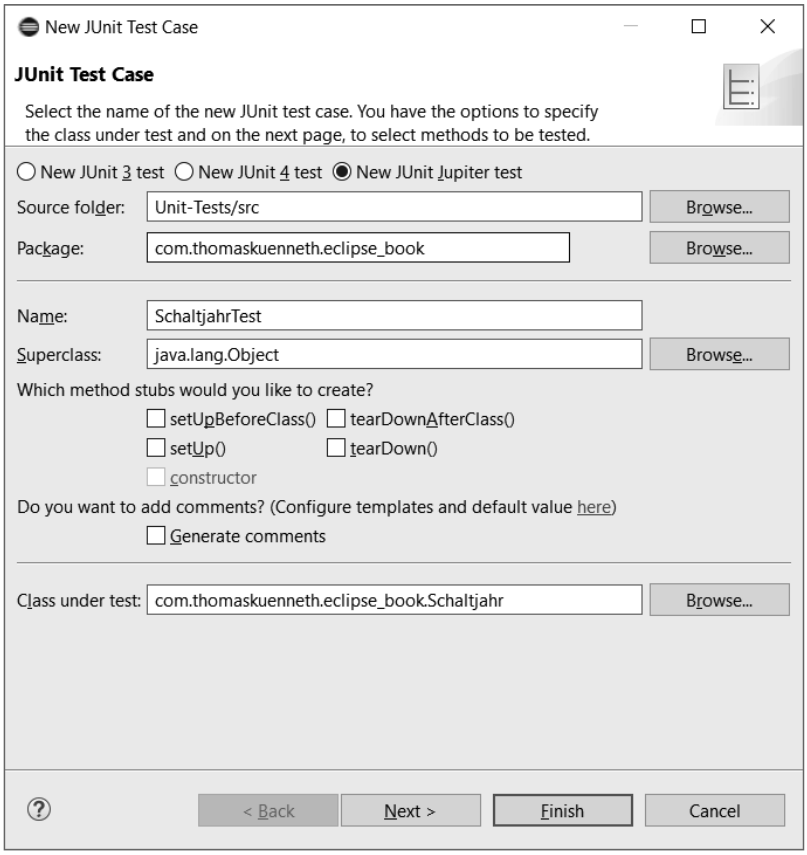

Bild 4.30 Assistent zum Anlegen eines neuen Testfalls

Im oberen Bereich können Sie festlegen, für welche Version von JUnit Sie den Testfall anlegen möchten. Für dieses Beispiel selektieren Sie bitte **New JUnit Jupiter test**. Sollte im unteren Bereich des Dialogs die Warnung „JUnit 5 requires a Java 8 project“ angezeigt werden, klicken Sie bitte auf **Configure**, um die Projekteinstellungen entsprechend anzupassen. Die übrigen Einstellungen, beispielsweise den Namen des Testfalls und die zu erzeugenden Methoden, können Sie unverändert übernehmen. Mit **Next** gelangen Sie auf die in Bild 4.31 gezeigte zweite Seite Test Methods des Assistenten zum Anlegen von Testfällen.

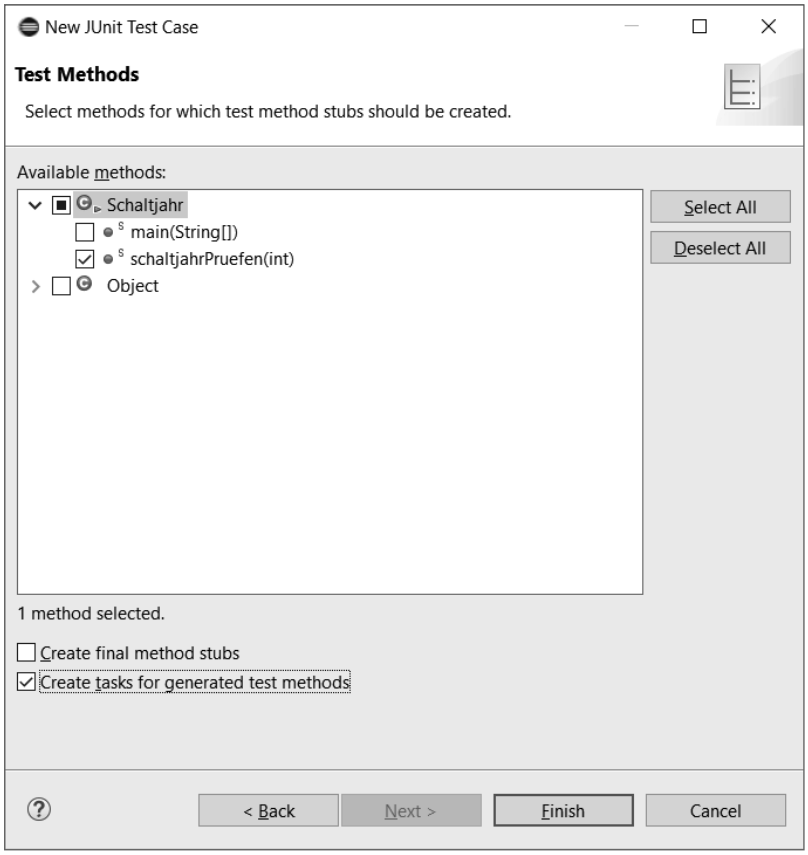

Bild 4.31 Die zweite Seite des Assistenten zum Anlegen von Testfällen

Hier können Sie diejenigen Methoden der zu testenden Klasse auswählen, für die Eclipse entsprechende Testmethoden generieren soll. `Schaltjahr` besteht nur aus zwei Methoden, wobei `main()` den Eingabeparameter entgegennimmt, die eigentliche Methode zum Ermitteln des Schaltjahrs aufruft und eine Ausgabe auf die Konsole liefert. Setzen Sie deshalb nur ein Häkchen vor `schaltjahrPruefen()`. Stellen Sie aber auch sicher, dass **Create tasks for generated test methods** im unteren Bereich des Dialogs ebenfalls markiert ist. Auf diese Weise können Sie in der Sicht *Tasks* nämlich sehr leicht feststellen, welche Testmethoden Sie noch implementieren müssen. Schließen Sie im Anschluss daran den Assistenten mit **Finish**. Da Sie Ihrem aktuellen Projekt noch keinen Testfall hinzugefügt haben, weist Sie Eclipse mit dem in Bild 4.32 gezeigten Dialog darauf hin, dass sich JUnit 5 nicht im Build Path befindet. Stimmen Sie dem Vorschlag (**Perform the following action: Add JUnit 5 library to the build path**) zu und beenden Sie den Dialog mit **OK**.

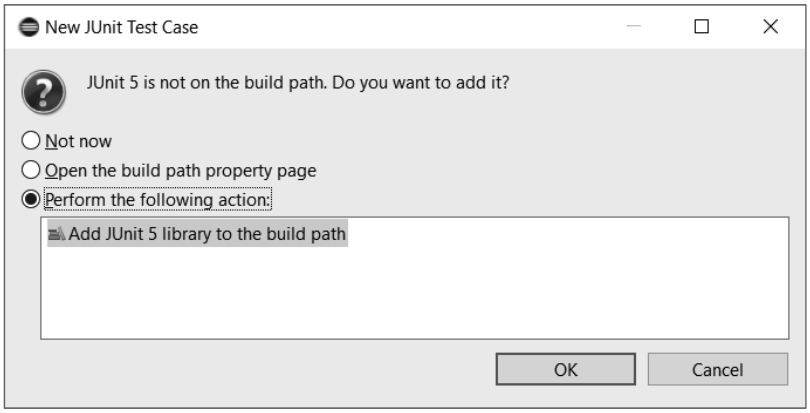

Bild 4.32 Hinweis, dass sich JUnit 5 nicht im Build Path befindet

Die generierte Klasse `SchaltjahrTest` wird im Package Explorer angezeigt und im Java-Editor geöffnet. Die Prüfmethode `testSchaltjahrPruefen()` enthält im Moment nur eine Anweisung, nämlich den Aufruf der Methode `fail()`. Sie muss also noch für den eigentlichen Test vorbereitet werden.

HINWEIS: Wenn im Java-Editor Klassen, die zu JUnit gehören, nicht gefunden werden, lohnt ein Blick auf die Registerkarte **Libraries** der Seite *Java Build Path* in den Projekteigenschaften. Klappen Sie die beiden Knoten **Modulepath** und **Classpath** auf. Danach können Sie den JUnit-Eintrag mittels Drag and Drop in den **Modulepath**-Zweig verschieben. Nach dem Hinzufügen von fehlenden Imports und dem erneuten Bauen des Projekts sind die Fehler verschwunden.

Eine Testmethode implementieren

In Testmethoden werden Annahmen formuliert und mit den Ergebnissen von Methodenaufrufen (der zu testenden Klasse) verglichen. Um zu prüfen, ob `schaltjahrPruefen()` richtig funktioniert, könnten Sie folgende Hypothese aufstellen: Sie ist korrekt implementiert, falls sie sowohl bei Schaltjahren als auch bei allen übrigen Jahreszahlen das richtige Ergebnis liefert. In JUnit werden solche Annahmen mithilfe der Methoden `assert...()` formuliert. Wenn Sie beispielsweise `assertFalse()` aufrufen, nehmen Sie an, dass eine Bedingung nicht erfüllt ist. 1901 war kein Schaltjahr. Also müsste `schaltjahrPruefen()` den Wert `false` liefern. Hieraus ergibt sich in der Java-Syntax folgende Annahme: `assertFalse(schaltjahrPruefen(1901));`. Um einen ersten Testlauf zu starten, übernehmen Sie bitte den folgenden Quelltext anstelle der ursprünglichen Implementierung der Klasse `SchaltjahrTest`:

Listing 4.14 SchaltjahrTest.java

```
package com.thomaskuenneth.eclipse_book;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
```

```
import org.junit.jupiter.api.Test;

class SchaltjahrTest {

  @Test
  public void testSchaltjahrPruefen() {
    int jahr;
    boolean ergebnis;
    // auf "ist kein Schaltjahr" prüfen
    jahr = 1903;
    ergebnis = Schaltjahr.schaltjahrPruefen(jahr);
    assertFalse(jahr + " ist kein Schaltjahr", ergebnis);
    // auf "ist ein Schaltjahr" prüfen
    jahr = 1904;
    ergebnis = Schaltjahr.schaltjahrPruefen(jahr);
    assertTrue(jahr + " ist ein Schaltjahr", ergebnis);
  }
}
```

Die Testmethode formuliert zwei Annahmen, nämlich dass 1904 ein Schaltjahr war, 1903 hingegen nicht.

Der erste Testlauf

Starten Sie nun mit **Run/Run As/JUnit Test** einen Testlauf. Die Ergebnisse von JUnit-Läufen werden in der Sicht *JUnit* angezeigt, die Sie in Bild 4.33 sehen. Falls Eclipse sie nicht automatisch geöffnet hat, können Sie dies jederzeit über **Window/Show View/Other** nachholen. Sie finden die Sicht unterhalb des Knotens **Java**. Wenn Sie möchten, können Sie den Namen auch in der Filterzeile des Dialogs eintippen.

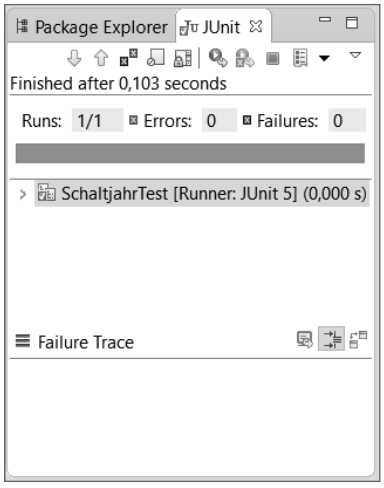

Bild 4.33
Die Sicht „JUnit“

Auffallend ist der charakteristische Balken im oberen Bereich der Sicht, der das Ergebnis eines Testlaufs anzeigt. Die Farbe Grün signalisiert, dass die Methode `schaltjahrPruefen()` für die beiden Annahmen des Testfalls die richtigen Werte geliefert hat. Allerdings wäre es voreilig, nur aufgrund zweier richtiger Annahmen auf die Korrektheit der Implementierung zu schließen.

Im gregorianischen Kalender werden die Schaltjahre nach den folgenden drei Regeln ermittelt:

- Alle ohne Rest durch 4 teilbaren Jahre sind Schaltjahre.
- Alle ohne Rest durch 100 teilbaren Jahre sind keine Schaltjahre.
- Alle ohne Rest durch 400 teilbaren Jahre sind Schaltjahre.

Die erste Regel wird im Testfall geprüft. Um auch das Einhalten der beiden anderen Bedingungen zu verifizieren, fügen Sie bitte die folgenden Annahmen hinzu:

```
assertFalse(Schaltjahr.schaltjahrPruefen(1900));
assertTrue(Schaltjahr.schaltjahrPruefen(2000));
```

Das Jahr 1900 war nämlich aufgrund der 100er-Regel kein Schaltjahr, 2000 wegen seiner Teilbarkeit durch 400 hingegen schon, auch wenn hier ebenfalls die 100er-Regel greifen würde.

Starten Sie den Testlauf erneut. Die Sicht *JUnit* zeigt nun einen roten Balken (Bild 4.34). Unter **Failure Trace** können Sie sehen, welcher Fehler aufgetreten ist. Die Annahme, dass 1900 kein Schaltjahr ist, wurde nicht erfüllt. Die Methode `schaltjahrPruefen()` hat hier also fälschlicherweise `true` geliefert. Die Erklärung für dieses Verhalten ist natürlich einfach. Die Anweisung, die die Variable `schaltjahr` auf `false` setzen würde, war auskommentiert.

Bild 4.34 Fehler bei der Abarbeitung des Testfalls

Wenn Sie möchten, können Sie die Kommentarzeichen // aus der Zeile `// schaltjahr = false;` in der Klasse `Schaltjahr` entfernen und anschließend den Testlauf erneut starten. Klicken Sie hierzu einfach auf das Symbol (**Rerun Test**) der Sicht *JUnit*. Der Balken ist wieder grün.

4.4.2 Weitere JUnit-Funktionen

Gerade bei umfangreichen Projekten werden sehr viele Testfälle benötigt. Um hier den Überblick zu behalten, sollten Sie die Tipps in diesem Abschnitt bei der Entwicklung im Hinterkopf behalten.

Den angezeigten Namen konfigurieren

Wenn Sie in der Sicht *JUnit* anstelle des Namens der Testmethode einen frei konfigurierbaren Text anzeigen möchten, stellen Sie der Methode einfach die Annotation `@DisplayName("…")` voran. Wie dies zur Ausführungszeit des Tests aussieht, ist in Bild 4.35 dargestellt. Es zeigt auch ein Kontextmenü, das Sie durch Anklicken der Testmethode in der Sicht *JUnit* mit der rechten Maustaste aufrufen können. **Go to File** bringt Sie direkt zu der Methode im Java-Editor.

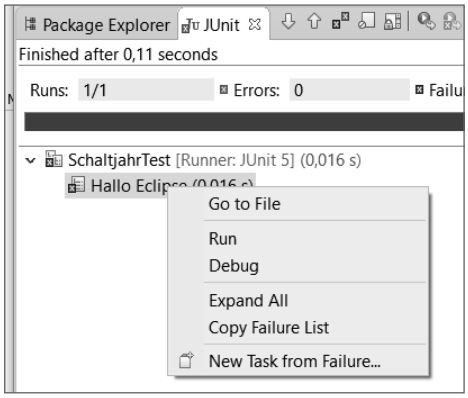

Bild 4.35 Kontextmenü einer Testmethode

Konfigurationen

Sie können mithilfe von Konfigurationen Einfluss auf das Laufzeitverhalten von Testfällen nehmen, wie Sie es auch vom Starten und Debuggen von normalen Java-Programmen her gewohnt sind. Klicken Sie hierzu die gewünschte Datei im Package Explorer mit der rechten Maustaste an und wählen Sie dann **Properties**. Auf der in Bild 4.36 gezeigten Seite Run/Debug Settings des Eigenschaften-Dialogs sehen Sie eine Liste der Konfigurationen, die dieser Datei zugeordnet sind.

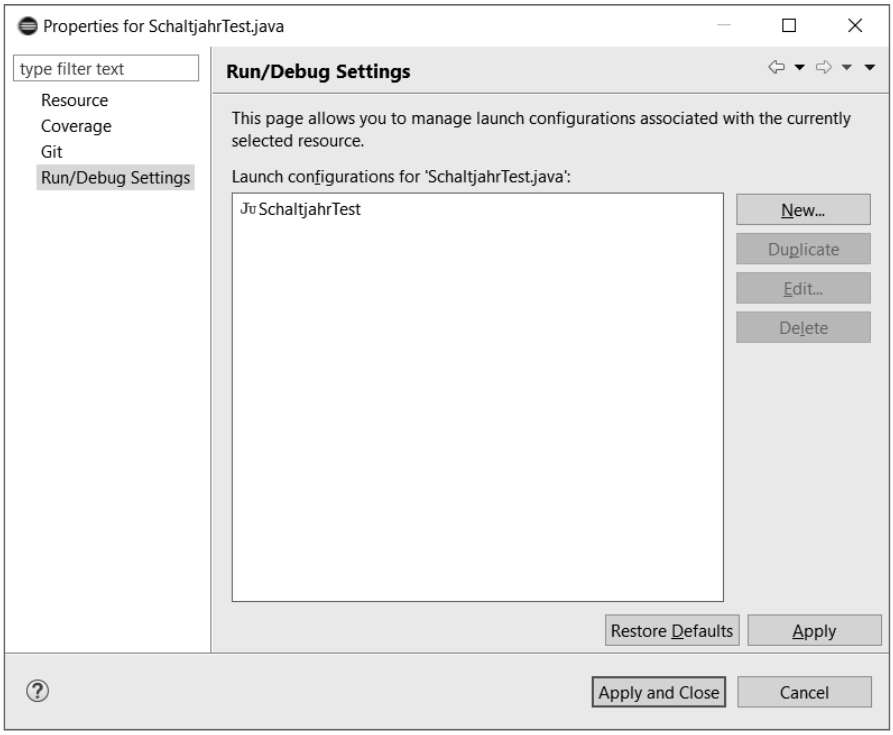

Bild 4.36 Die Seite „Run/Debug Settings“

Sie können bestehende JUnit-Konfigurationen bearbeiten sowie neue anlegen. In beiden Fällen sehen Sie den in Bild 4.37 gezeigten Konfigurationsdialog, auf dessen Registerkarten Sie das Laufzeitverhalten des Testfalls beeinflussen können. Beispielsweise stellen Sie auf der Karte **Test** die zu verwendende JUnit-Version ein.

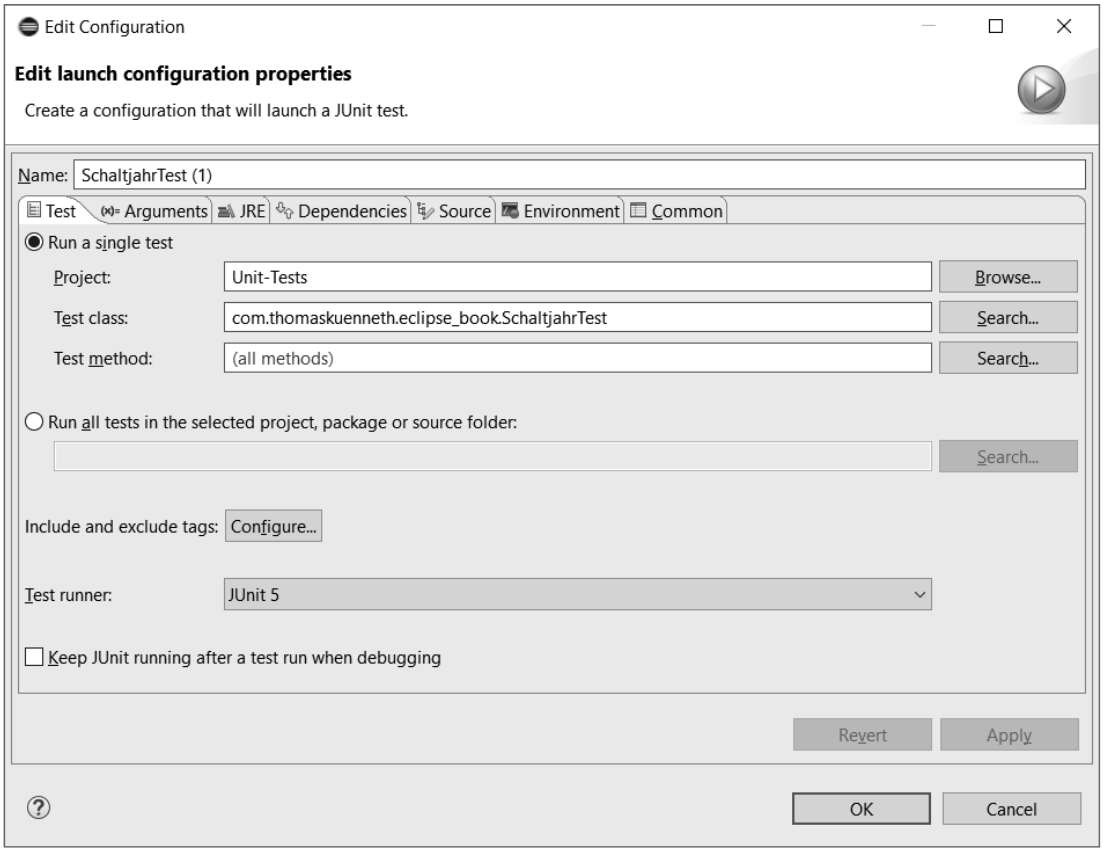

Bild 4.37 Bearbeiten einer JUnit-Konfiguration

4.5 Zusammenfassung

JUnit ist ein wichtiges Werkzeug in der Softwareentwicklung, das bei richtiger Anwendung nicht nur beim Aufdecken von offensichtlichen, sondern auch bei mit herkömmlichen Mitteln praktisch nicht zu findenden Fehlern helfen kann. Wie Sie an diesem kleinen Beispiel gesehen haben, kann das Framework aber nur funktionieren, wenn die Testfälle mögliche Problemfelder auch wirklich aufdecken können. Um ein Gespür für das Schreiben von Tests zu bekommen, rate ich Ihnen deshalb zu entsprechender weiterführender Literatur. Sie finden sie in der Lektüreliste im Anhang dieses Buchs.

Auch wenn sich Unit-Tests zu einem fundamentalen Bestandteil des Softwareentwicklungsprozesses gemausert haben, können und sollen sie aber das klassische Debuggen nicht ersetzen. Vielmehr erweitern sie das Repertoire des Entwicklers. Unit-Tests werden idealerweise frühzeitig eingesetzt und über den kompletten Entwicklungszeitraum kontinuierlich wiederholt. Eclipse unterstützt Sie hierbei nicht nur beim Anlegen von Testfällen, sondern auch bei deren Ausführung.