Java

Geschichte

1992 Urversion von Java mit dem Namen Oak entsteht im Auftrag von Sun Microsystems

1995 Offizielle Vorstellung von Java

Grundkonzepte

Objektorientierte Programmiersprache

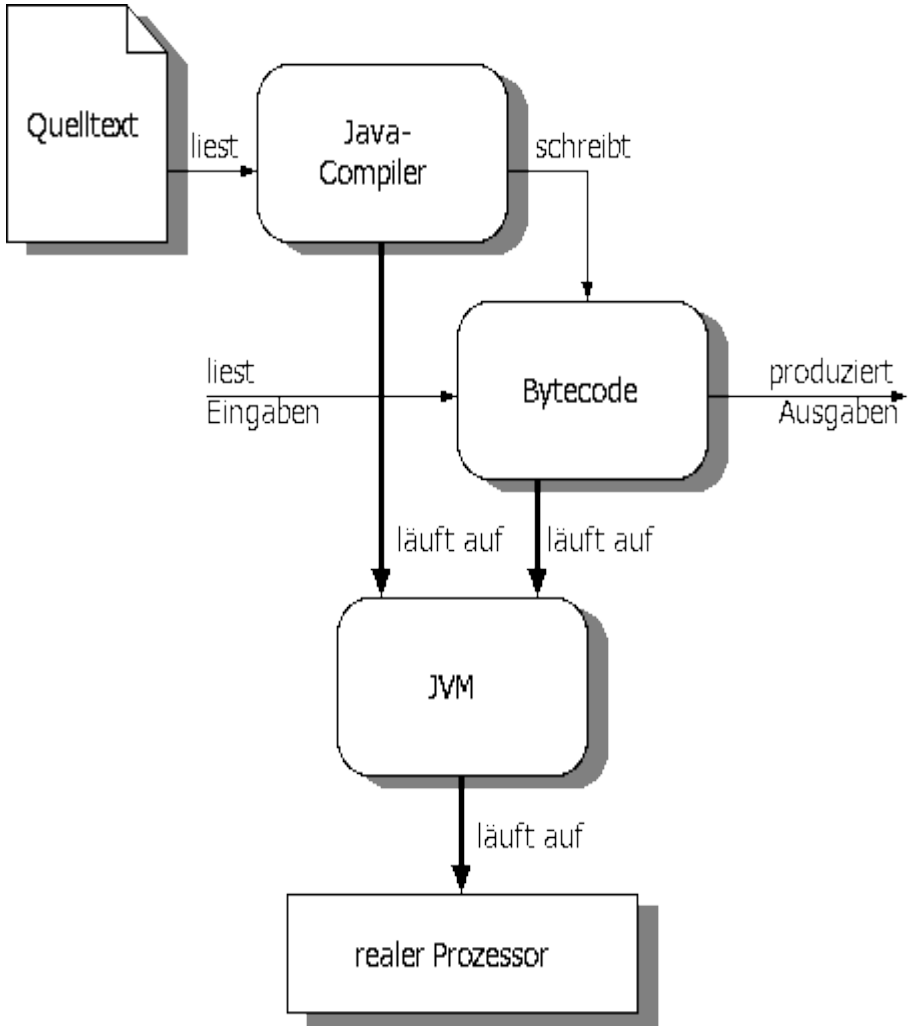
Portabilität/Plattformunabhängigkeit

Anlehnung an bekannte Programmiersprachen wie C++

Eingebaute Unterstützung für die Verwendung von Computernetzen

Plattformunabhängigkeit

- Javacompiler erzeugt Bytecode für die virutelle Maschine(JVM)
- Bytecode wird von der JVM interpretiert
- Java-Programme laufen auf allen Systemen, für die es eine JVM gibt
- Java Runtime Enviroment(JRE) verfügbar auf www.sun.com



Objektorientierung

Grundidee: Abbildung von realen Objekten

Trägt zur Flexibilität und Wiederverwendbarkeit von Programmen bei

Jedes Objekt hat individuelle Attribute

Aber: Java nicht vollständig objektorientiert

Die primitiven Datentypen sind keine Objekte

Klassen

Muster für strukturierte Datentypen und darauf definierten Operationen

Aus Klassen werden Objekte erzeugt

Bestandteile:

Variablen

Methoden (Main-Methode)

Konstruktoren

Beispielklasse

```
class Sum
{
   public static void main(String... args)  //Mainmethode
   {
     int n;              /*
     int s;                Variablen definieren
     int i;              */

     n = 4;              /*
     s = 0;                Wertzuweisung
     i = 1;              */
     while(i <= n)
     {
        s = s + i;
        i = i + 1;
     }
     System.out.println(s);
   }
}
```

Primitive Datentypen

Name	Kleinster Wert	Größter Wert	Größe in Bit
byte	-127	128	8
short	32.768	32.767	16
int	-2.147.483.648	2.147.483.647	32
long	-9.223.372.036.854.775.808	9.223.372.036.854.775.807	64
char	\u0000 (0)	\uFFFF (65.535)	16 (Unicode)
boolean	false		
float	ca. 1,40e-45 *	3,40282346638528860e+38	32 (IEEE 754)
double	4,94065645841246544e-324	1,79769313486231570e-308	

Konvention zur Namensgebung

Variablen, Methoden, primitive Typen

erster Buchstabe klein, nachfolgende Wortteile groß: (counter, findBy1stToken, bottomUp)

Referenztypen

erster Buchstabe groß, nachfolgende Wortteile groß: (Sum, String, Rational)

Variablen mit sehr kurzem Gültigkeitsbereich

einzelne kleine Buchstaben

Typvariablen in Generics

einzelne große Buchstaben

statische, öffentliche Konstanten

- alle Buchstaben groß, Wortteile getrennt mit Underscore (MAX_VALUE, PI, RGB24)

Definition von Variablen

- Variable definieren: int i;
- Variable initialisieren: i = 1; alternativ: int i = 1;

Wichtig: Nie uninitialisierte Varaiblen

Operatoren

Die wesentlichen Operatoren aus anderen Programmiersprachen sind auch in Java vorhanden, wie z.B.:

+,-,~(bitweise neg.),*,/,%(modulo),+,-,<,<=,>,>=

Die bekannten Kontroll- und Schleifenkonstruktionen sind ebenfalls vorhanden:

If, while, do while, for, switch case

Methoden

Methoden beschreiben verhalten von Objekten

Methodendefinition:

Modifier: gibt zugriffsrechte an

Mit Rückgabewert: Typ des Rückgabewertes

Ohne Rückgabewert: void

Indentifier: Name der Methode

Evtl. : Parameter

Methodenrumpf: in geschweiften Klammern die Anweisungen

Beispiel

```
public class Fact
{
    public static int factorial(int n)  /*Methode factorial mit modifier public, static
    {                                        Rückgabewert int und Parameter n*/
        int i = 1;
        int s = 1;
        while(i <= n)
        {
            s= s * i;
            i = i +1;
        }
        return s;
    }
    public static void main(String... args)
    {
        System.out.println(factorial(6));   //Methodenaufruf mit dem Parameter 6
    }
}
```

Konstruktoren

Konstruktoren dienen der Initialisierung von Datenfeldern bei der Erzeugung von Objekten

- ist kein Konstruktor in einer Klasse angegeben, so wird ein default-Konstruktor vom Java-Compiler generiert.
- In diesem werden alle Datenkomponenten mit den default-Werten initialisiert.
- Namenskonvention: Konstruktorname = Klassenname
- Konstruktoren können mit Parametern versehen werden
- Aufruf erfolgt mittels new Name(Parameter);

Beispiel

```
class X
{
  private
  int i;
  public X(int i)
{
    this.i = i;
  }
}

X test = new X(9);  //erzeugt Objekt von Typ X mit Parameter 9
```

Vererbung

Klassen in Java hierarchisch angeordnet

Jede neu definierte Klasse erweitert eine andere

Unterklassen erben die Eigenschaften und Methoden der Oberklasse

Alle Klassen in Java sind von der Klasse „Object“ abgeleitet

In Java keine Mehrfachvererbung wie in C++, d.h.: jede Klasse kann nur von einer Oberklasse erben

z.B.: class Test extends object{ ...}

Interfaces

Ein Klasse kann beliebig viele Interfaces implementieren

Module mit gleicher Schnittelle können gegeneinander ausgetauscht werden

Ein Interface gibt an, welche Methoden in einer Klasse vorhanden sind

Dies dient der Modularisierung der Softwarearchitektur

Beispiel

```
interface Complex
{
    ...
    public Complex add(Complex other);
    public Complex mult(Complex other);
}

class Cartesian implements Complex
{
    public Complex add(Complex other);
    {...}
    public Complex mult(Complex other);
    {...}
}
```

Packages

Programme, bestehend aus mehreren Klassen lassen sich zu Packages zusammenfassen

Package Name steht am Klassenanfang

Gängiges Packageformat: *.jar

Es gibt bereits viele vorgefertigte Klassen und Packages, die sich per import-Anweisung einbinden lassen, wie z.B. java.util.math

Exceptions

Exceptions dienen dem Umgang mit zur Laufzeit auftretenden Fehlern

- Tritt beim Ausführen eines Programmblockes ein Fehler auf, kann eine entsprechende Exception generiert werden und der Fehler kann dann abgefangen und behandelt werden
- Exceptions werden mit throws exception und throw new exception erzeugt
- Mit try und catch lassen sich exception behandeln
- Es gibt auch die Klasse vordefinierte Exception-Klasse, welche meistens jedoch zu unspezifisch ist und angepast werden muss

JUnit Überblick

- Was ist JUnit?
- Allgemeine Struktur von JUnit
- Funktionen und Vorteile von JUnit
- Wie testet man mit JUnit?
- Ein kleines Beispiel für Tests

Was ist JUnit?

- Framework für die Durchführung von Black-Box-Tests
- Testfälle werden in eigenen Testklassen kodiert und zu TestSuiten zusammengefasst
- Strikte Trennung des Codes von Anwendung und Test sichergestellt.
- Einfachheit bei Implementierung und Analyse der Ergebnisse
- Schnelligkeit bei Ausführung von Tests
- Open Source

Allgemeine Struktur von JUnit

- Homepage:
  - http://www.junit.org
  - Freier Download über sourceforge.net
  - Aktuelle Version: JUnit 4.5
  - Bestandteile: JAR Archiv, Source Code, Api Dokumentation, Dokumentation zu JUnit

Allgemeine Struktur von JUnit

- Installation:
  - die **junit.jar** dem **CLASSPATH** hinzufügen
  - Alle zum Testen notwendigen Klassen sind im Paket **junit.framework** enthalten.
  - Weiteres Vorgehen: Zu jeder verfassten Klasse eine Testklasse entwerfen.
  - Zum Schreiben von Tests werden lediglich benötigt: **TestCase**, **Assert**, (**TestSuite**)
  - Zum Ausführen der Tests werden benötigt: **TestRunner**

Funktionen von JUnit

- TestCase, Assert und TestRunner
  - JUnit stellt die abstrakte Klasse **TestCase** zur Verfügung.
  - **TestCase** wird auf einen speziellen Testfall abgeleitet und erweitert.
  - Framework führt die abgeleiteten Testfälle aus.
  - **Assert**: eine Behauptung, dass das Ergebnis eines Tests ein bestimmtes Ergebnis zurückliefern sollte.
  - Der **TestRunner** führt den eigentlichen Test durch

JUnit: Die Idee

- Grundkonzepte: TestCases und TestSuites
  - Unit Test: Subklasse von **junit.framework.TestCase**
  - TestCases lassen sich zu einer größeren Einheit kombinieren, einer sog. TestSuite
  - TestSuites lassen sich innerhalb anderer TestSuites einbinden
  - TestRunner arbeitet die Tests (d.h TestCase oder TestSuite) nach einem einheitlichem Muster ab

Vorteile von JUnit

- Änderungen am Code können sofort überprüft werden
- Mehr Sicherheit bezüglich der Qualität des Codes
- Weniger Angst vor Änderungen an Kernkomponenten
- Entwicklungszeiten werden kürzer (trotz der zusätzlich zu implementierenden Tests)
- Einfacheres Überarbeiten (Refactoring) alter Codesegmente
- Weitere, nicht mit der Anwendung vertraute, Entwickler können sich schneller einarbeiten
- Der Code der Anwendung bleibt selbst bei steigender Komplexität übersichtlich

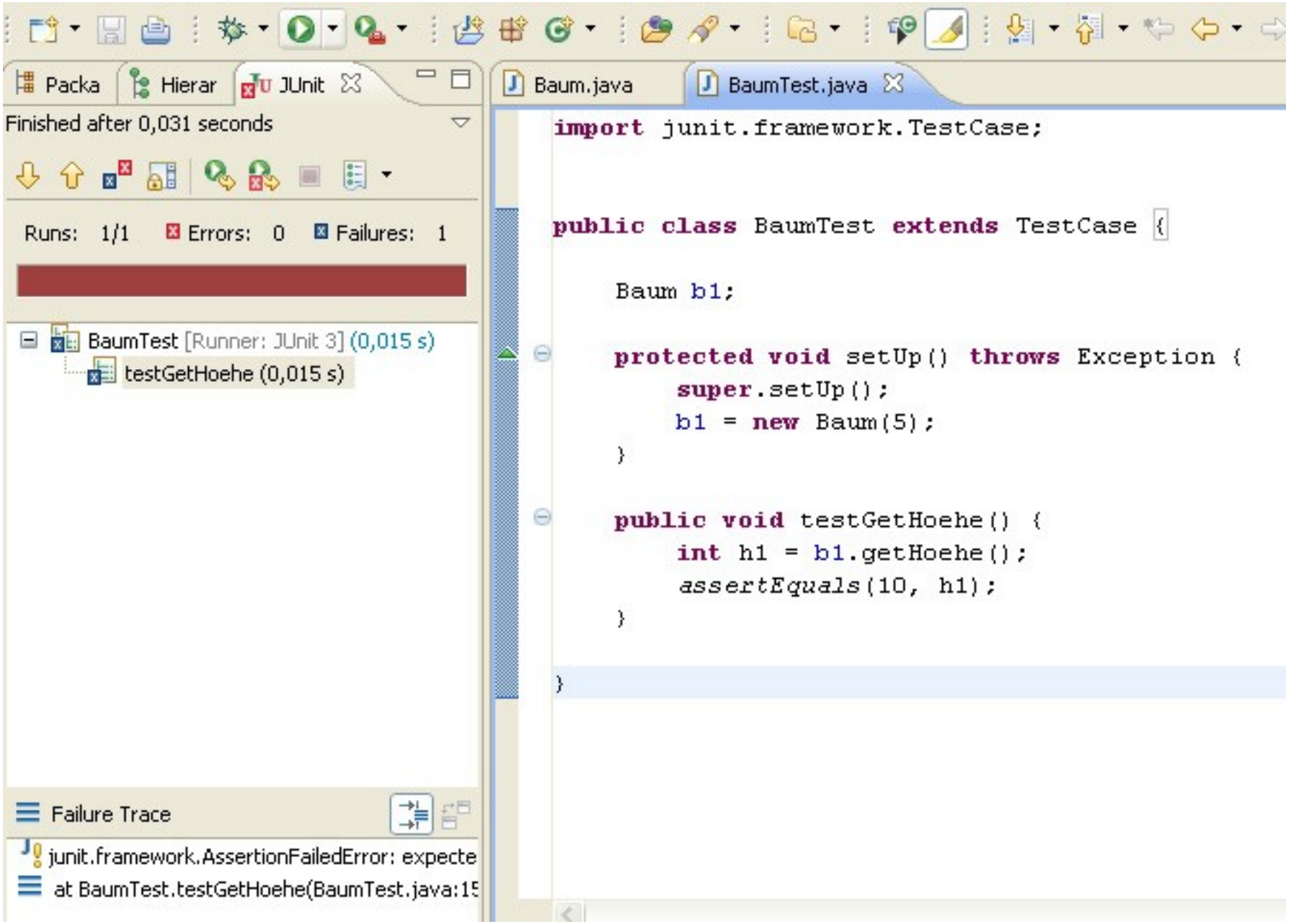
Wie testet man mit JUnit?

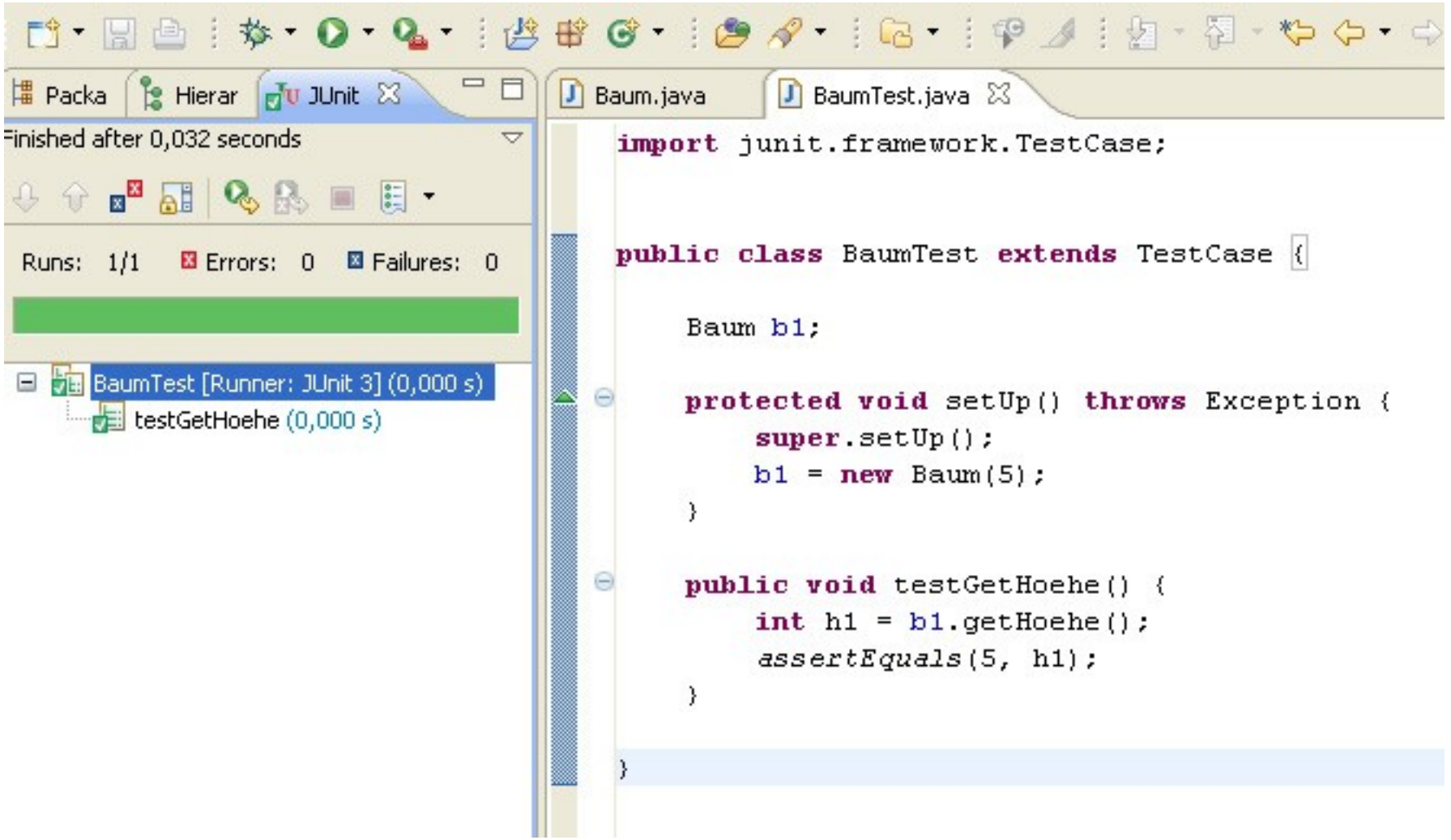
- Erstelle eine von TestCase abgeleitete Klasse
- Erstelle darin einen Constructor, der einen String als Eingabe erhält und an die Superclass übergibt (ab JUnit 3.8 nicht mehr nötig)
- Überschreibe die Methode runTest(), so dass darin einzelne Methoden aufgerufen werden
- Wenn ein Wert überprüft werden soll, muss die Methode assertTrue aufgerufen werden
- Weitere Vergleiche sind: assertFalse, assertEquals, assertArrayEquals
- Weitere Methoden sind: setUp, tearDown

Ein Beispiel

```
public class Baum {

    private int hoehe;

    public Baum(int hoehe) {
        this.hoehe = hoehe;
    }

    public int getHoehe() {
        return this.hoehe;
    }

    public void setHoehe(int hoehe) {
        this.hoehe = hoehe;
    }
}
```

```java
import junit.framework.TestCase;


public class BaumTest extends TestCase {

    Baum b1;

    protected void setUp() throws Exception {
        super.setUp();
        b1 = new Baum(5);
    }

    public void testGetHoehe() {
        int h1 = b1.getHoehe();
        assertEquals(10, h1);
    }

}
```


Packa
Hierar
JUnit
Finished after 0,031 seconds
Runs: 1/1
Errors: 0
Failures: 1
BaumTest [Runner: JUnit 3] (0,015 s)
testGetHoehe (0,015 s)
Failure Trace
junit.framework.AssertionFailedError: expecte
at BaumTest.testGetHoehe(BaumTest.java:15
Baum.java
BaumTest.java
import junit.framework.TestCase;
public class BaumTest extends TestCase {
Baum b1;
protected void setUp() throws Exception {
super.setUp();
b1 = new Baum(5);
}
public void testGetHoehe() {
int h1 = b1.getHoehe();
assertEquals(10, h1);
}
}


Packa
Hierar
JUnit
Finished after 0,032 seconds
Runs: 1/1
Errors: 0
Failures: 0
BaumTest [Runner: JUnit 3] (0,000 s)
testGetHoehe (0,000 s)
Baum.java
BaumTest.java
import junit.framework.TestCase;
public class BaumTest extends TestCase {
Baum b1;
protected void setUp() throws Exception {
super.setUp();
b1 = new Baum(5);
}
public void testGetHoehe() {
int h1 = b1.getHoehe();
assertEquals(5, h1);
}
}

log4j

Was ist log4j?

- Logging-API für Java
- Komfortable Weise der Verfolgung und Protokollierung von Programmabläufen
- Aufwandsreduktion bei der Fehlersuche

Homepage zum Download:
- http://logging.apache.org/log4j

log4j

Komponenten

- Logger
- Appender
- Layout

log4j

Logger

- Logger benutzen verschiedene Appender und Layouts
- Ein Logger kann beliebig viele Appender haben
- Logger benutzen verschiedene Levels für die Ausgabe
  - TRACE< DEBUG < INFO < WARN < ERROR < FATAL
- Logger können pro Klasse definiert werden

log4j

- Appender

  - Appender definiert das Ausgabe-Medium eines Loggers
    z.B. Datei, Konsole, DB, Auswertung, etc..
  - Jedem Appender muss ein Layout zugeordnet werden

- Layout

  - Definiert die Darstellung der Protokollierung auf dem Ausgabe-Medium
    z.B. Layout für eine Datei oder Console

log4j

- Log4J erste Schritte in Eclipse
  - Download unter : http://logging.apache.org/log4j
  - Neues Javaprojekt in Eclipse erstellen
  - die log4j.jar in dem Build-Path vom Projekt importieren

log4j

- Neue Java-Klasse erstellen
  folgenden Text kopieren :

```
import org.apache.log4j.Logger;
public class log4jTest {
    private static  org.apache.log4j.Logger log =
    Logger.getLogger(log4jTest.class);
    public static void main(String[] args) {
        log.trace("Trace");
        log.debug("Debug");
        log.info("Info");
        log.warn("Warn");
        log.error("Error");
        log.fatal("Fatal");
    }
}
```

log4j

- Klasse speichern
- Eine neue Textdatei Namens „log4j.properties“ im src-Verzeichnis anlegen und den folgenden Inhalt kopieren :

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=
    %d{ABSOLUTE} %5p %c{1}:%L - %m%n
log4j.rootLogger=debug, stdout
```

log4j

- Programm ausführen und Ausgabe kontrollieren
  Die Ausgabe sollte wie folgt aussehen :
  08:50:49,661 DEBUG LogClass:29 - Debug
  08:50:49,663 INFO LogClass:30 – Info
  08:50:49,663 WARN LogClass:31 - Warn
  08:50:49,663 ERROR LogClass:32 – Error
  08:50:49,664 FATAL LogClass:33 - Fatal
- Ändere nun die letzte Zeile
  log4j.rootLogger=debug, stdout
  auf „log4j.rootLogger=warn, stdout“ und führe das Programm neu aus.
  Es werden jetzt nur noch Levels die größer gleich „warn“ sind, geloggt.

log4j

- Beispiel für ein RollingFileAppender
  Ändere die log4j.properties wie folgt :

```
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.maxFileSize=100KB
log4j.appender.file.maxBackupIndex=5
log4j.appender.file.File= c:\meinErstesLogger.log (z.B.)
log4j.appender.file.threshold=info
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE}
    %5p %c{1}:%L - %m%n
log4j.rootLogger=„Level“ , file
```

- Weitere Hinweise und Tipps v.a. zu Appendern :

http://logging.apache.org/log4j/docs/api/org/apache/log4j/AppenderSkeleton.html

Javadoc

- Was ist Javadoc?
  - Ein Werkzeug zum automatischen Generieren von HTML-Dokumentationen aus Java-Quell-Code

- Vorteil :
  - Dokumentation findet während der Programmierung statt
  - Code und Dokumentation können im Quelltext stehen

Javadoc

- Block Tags (für Klassen, Methoden oder Attribute)
  - @author <name>
    Name des Autors
  - @version <id>
    Versionsbezeichnung
  - @param <name> <bedeutung>
    Parametername & Bedeutung
  - @return <bedeutung>
    Erläuterung desRückgabewertes
  - @see <querverweis>
    Schafft einen Querverweis auf
    einen anderen dokumentierten Namen

Javadoc

- In-Line Tags
  - Können in den Beschreibungs Kommentaren von Klassen, Methoden oder Block Tags stehen.
- {@link Class#method}
- HTML Textformatierungen

Javadoc

Beispiel für Javadoc mit dem Quell-Code aus der Klasse Logger :

```
package org.apache.log4j:

import org.apache.log4j.spi.LoggerFactory;
import org.apache.log4j.Level;
/**  This is the central class in the log4j package. Most logging
  operations, except configuration, are done through this class.
 @since log4j 1.2
 @author Ceki G&uuml;lc&uuml; */
public class Logger extends Category {
  /**   The fully qualified name of the Logger class. See also the
     getFQCN method. */
  private static final String FQCN = Logger.class.getName();
  protected
  Logger(String name) {
    super(name);
  }
```

org.apache.log4j

Class Logger

```
java.lang.Object
  └Category
      └org.apache.log4j.Logger
```

```
public class Logger
extends Category
```

This is the central class in the log4j package. Most logging operations, except configuration, are done through this class.

Since:

log4j 1.2

Constructor Summary

protected	Logger(java.lang.String name)

Method Summary

static Logger	getLogger(java.lang.Class clazz) Shorthand for getLogger(clazz.getName()).
static Logger	getLogger(java.lang.String name) Retrieve a logger named according to the value of the name parameter.
static Logger	getLogger(java.lang.String name, LoggerFactory factory)