

Exceptions

Exceptions / Ausnahmen

- Eine Ausnahme verhindert den normalen Programmablauf
- Aufgabe 19: Nennen Sie mögliche Ursachen für eine Ausnahme!
- Programm muss auch in einer Ausnahmesituation stabil sein
 - kein unkontrollierter Abbruch
 - keine fehlerhafte Berechnung
 - kein Datenverlust
- Wir benötigen einen Informationskanal, um eine Ausnahmesituation zu signalisieren

- Idee: Rückgabewert einer Methode als Fehlersignal
 - Beispiel:

```
public class Manager extends Angestellter{
    private double bonus;

    // ...

    public boolean setBonus(double b){
        if (b >= 0){
            bonus = b;
            return true;
        } else {
            return false;
        }
    }
}
```

- Nachteile:

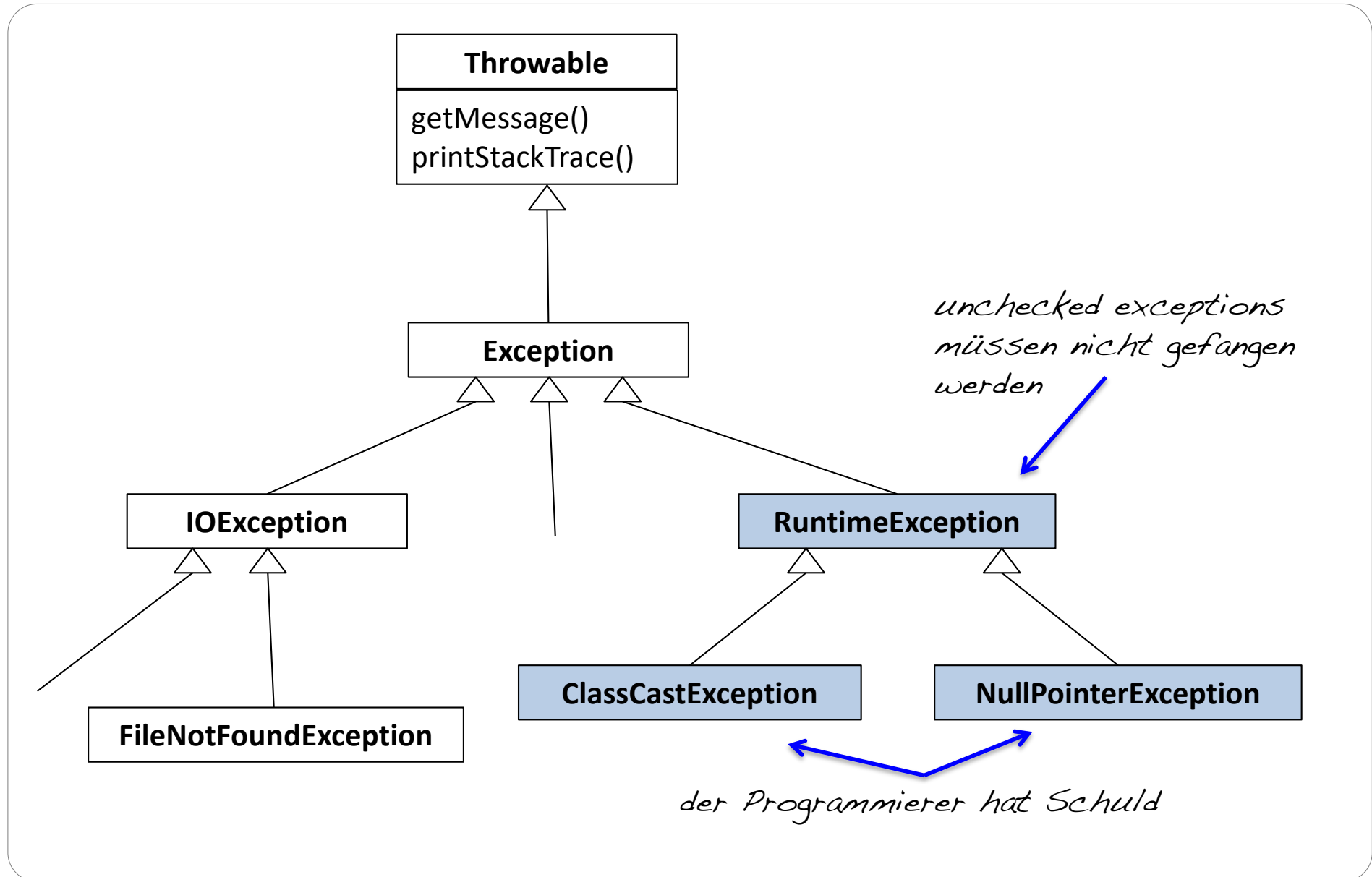
- der Aufrufer darf die Fehlerbehandlung nicht vergessen
(keine Prüfung durch Compiler möglich)

```
public void verwalteBonus(Manager m, double bonus){  
    boolean isOK = m.setBonus(bonus);  
    if (!isOK){  
        System.out.println("Bitte Bonus korrigieren");  
        // Ausnahmebehandlung  
    }  
    //  
}
```

- der Aufrufer muss das Protokoll kennen
(hier: Rückgabewert `false` bei Fehler)
- Wenn die Methode auch einen fachlichen Rückgabewert liefern soll, wird das Protokoll komplexer, bzw. der Rückgabewert kann nicht mehr als Fehlersignal verwendet werden (wenn der Wertebereich zu klein ist)
- Ausnahmebehandlung kann nicht erzwungen werden
- Keine Trennung zwischen fachlichem Code und Ausnahmebehandlung

- Eine „manuelle“ Ausnahmebehandlung hat erhebliche Nachteile
- Besser: Sprache bietet einen Mechanismus für eine geregelte Ausnahmebehandlung
- Java bietet *Exceptions*
 - Eigene Datenstruktur für Ausnahmesignale (Klasse `Exception`)
 - Eigener Kanal zur Signalisierung von Ausnahmen (`throw`)
 - Trennung der Ausnahmebehandlung vom regulären Ablauf (`try-catch`)
 - Compiler kann auf fehlende Ausnahmebehandlung prüfen
 - Bestimmte Ausnahmen müssen zur Kenntnis genommen werden (vom Compiler erzwungen)

- Falls Fehler zur Laufzeit auftreten, bricht ein Java-Programm nicht sofort ab
 - Es wird zunächst eine **Exception** (Ausnahme) erzeugt/geworfen
 - Bei einer Exception handelt es sich um ein Objekt, welches einen Fehler repräsentiert
 - Mit einer **try-catch**-Anweisung kann man ein Ausnahme-Objekt „fangen“ und bearbeiten
 - Das System zur Fehlerbehandlung sucht dann in der Aufrufhierarchie nach einer Stelle, die den aufgetretenen Fehler bearbeiten kann
- Eine Ausnahme ist ein Objekt vom Typ `Throwable` (oder einer Unterklasse)
- In der Java-API (`java.lang`) sind bereits zahlreiche Ausnahmen definiert



– Einige bekannte RuntimeExceptions

Unterklasse von RuntimeException	Ausgelöst durch
ArithmeticException	Division durch 0
ArrayIndexOutOfBoundsException	Missachtung von Arraygrenzen
ClassCastException	unpassende Typkonvertierung zur Laufzeit
IllegalArgumentException	falsche Methodenargumente
NumberFormatException	unpassende Umwandlung
NullPointerException	Methodenaufruf auf null-Referenz
UnsupportedOperationException	Aufruf einer nicht gestatteten Operation

Definition eigener Ausnahmetypen

- Falls man in der Java-API keine passende `Exception`-Klasse findet, kann man einen neuen `Exception`-Typ durch Erweiterung einer bestehenden `Exception`-Klasse erstellen

```
public class BonusException extends Exception{  
    public BonusException(){  
        super();  
    }  
    public BonusException(String message){  
        super(message);  
    }  
}
```

eine bestehende Ausnahme wird erweitert

*mit diesem Konstruktor können
Informationen über die Ausnahme übergeben werden*

Werfen (Auslösen) einer Exception

- Mit **throw können** wir nun im Ausnahmefall ein entsprechendes Ausnahmeobjekt werfen
- Eine Methode, in der eine Ausnahme geworfen werden kann, muss dies in der Signatur mit **throws** anzeigen (gilt nicht für `RuntimeException`)

```
public void setBonus(double b) throws BonusException {  
    if (b >= 0) {  
        bonus = b;  
    } else {  
        throw new BonusException("Bonus ist negativ.");  
    }  
}
```

*die Methode muss bekanntgeben, dass
sie eine Ausnahme auslösen kann*

hier wird das Exception-Objekt „geworfen“

*ein entsprechendes Ausnahme-Objekt
wird erzeugt*

Fangen/Weiterleiten einer Exception

- Wenn wir eine Methode aufrufen, die eine geprüfte Ausnahme auslösen kann, haben wir zwei Möglichkeiten:

1) Verwenden einer try-catch-Anweisung

```
try {  
    Code  
    Methodenaufruf // kann Ausnahme auslösen  
    Code  
} catch (ExceptionType e) {  
    //Fehlerbehebung
```

*wird nur ausgeführt, falls
Ausnahme vom Typ ExceptionType
ausgelöst wird*

2) Weiterleiten der Exception mit throws

```
Typ Methodenname(...) throws ExceptionType {  
    //...  
    Methodenaufruf  
}
```

*Falls wir keine Möglichkeit haben, die
Ausnahme zu behandeln, geben wir sie
einfach weiter*

- Achtung: Das mögliche Auslösen einer RuntimeException muss nicht mit throws angezeigt werden!
- Ausnahmebehandlung:

```
public void verwalteBonus(Manager m, double bonus){  
    try {  
        m.setBonus(bonus);  
        System.out.println("Bonus wurde neu vergeben.");  
        //  
    } catch (BonusException e){  
        System.out.println("Bonus darf nicht negativ sein.");  
        System.err.println("Fehler: " + e.getMessage());  
        e.printStackTrace();  
        //  
    }  
}
```

wird nicht mehr ausgeführt, falls eine Ausnahme auftritt

optional

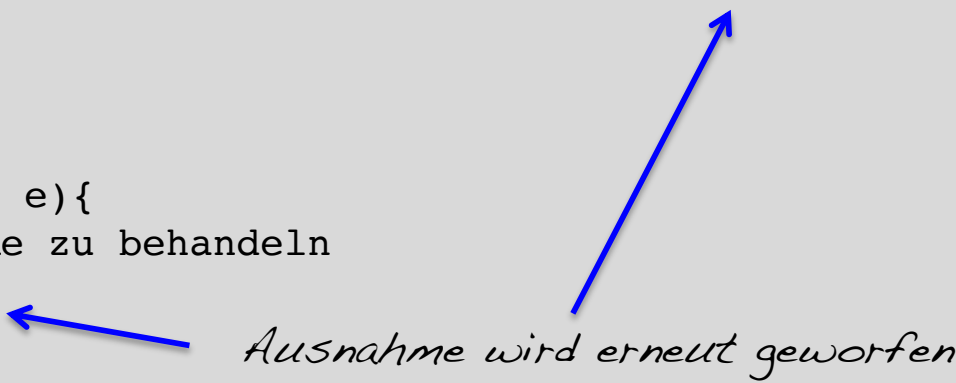
zeigt den Aufruf-Stack an

liefert Fehlerbeschreibung, die über den Konstruktor gesetzt wurde

Abbruch einer Ausnahmebehandlung

- Situation: Man hat ein Ausnahmeobjekt gefangen, stellt dann aber fest, dass man nicht in der Lage ist, die Ausnahme erfolgreich zu behandeln
 - Im catch-Block kann dasselbe Ausnahme-Objekt wieder geworfen werden, um die Ausnahme weiterzureichen.
 - Evtl. kann die Ausnahmesituation genauer bestimmt werden. Dann kann auch ein neues/spezielleres Ausnahme-Objekt erzeugt und geworfen werden

```
public void verwalteBonus(Manager m, double bonus) throws BonusException {  
    boolean noway = false;  
    try{  
        m.setBonus(bonus);  
        //  
    } catch (BonusException e){  
        // Probiere, Ausnahme zu behandeln  
        if (noway) throw e;  
    }  
}
```



The diagram illustrates the concept of re-throwing an exception. A blue arrow points from the `throw e;` statement in the catch block to the `throws BonusException` declaration in the method signature. Another blue arrow points from the handwritten text *Ausnahme wird erneut geworfen* to the `throw e;` statement.

Ausnahme wird erneut geworfen

finally-Block

- Falls ein bestimmter Code auf jeden Fall ausgeführt werden soll, egal ob eine Ausnahme aufgetreten ist oder nicht, kann der **finally**-Block verwendet werden

```
hardware.reservieren();  
try{  
    hardware.nutzen();  
    System.out.println("Hardware wurde genutzt.");  
} catch (HardwareException e){  
    System.out.println("Fehler bei Nutzung der Hardware.");  
} finally {  
    hardware.freigeben();  
    System.out.println("Hardware wieder frei.");  
}
```

wird auf jeden Fall ausgeführt

Mehrere catch-Blöcke

- Mit einem `try`-Block können unterschiedliche Ausnahmen gefangen werden

```
hardware.reservieren();  
try{  
    hardware.nutzen();  
    a[i] = 11;  
} catch (HardwareException e1){  
    // Fehlerbehandlung  
} catch (ArrayIndexOutOfBoundsException e2){  
    // Fehlerbehandlung  
} finally {  
    hardware.freigeben();  
}
```

- Eine Methode kann mehr als eine Exception auslösen. Die Ausnahmetypen werden, durch Kommata getrennt, nach `throws` aufgelistet

Exception-Hierarchie

- Da Ausnahmen Objekte sind, darf auch hier eine abgeleitete Klasse überall da stehen, wo die Basisklasse erlaubt ist
- Insbesondere kann man mit einer `Exception` auch eine `ArrayIndexOutOfBoundsException` fangen

```
hardware.reservieren();  
try{  
    hardware.nutzen();  
    a[i] = 11;  
} catch (Exception e1){  
    // Fehlerbehandlung  
} catch (ArrayIndexOutOfBoundsException e2){  
    // Fehlerbehandlung  
} finally {  
    hardware.freigeben();  
}
```

Compilefehler, da Exception bereits alle Ausnahmen fängt

die Basisklasse muss hinter der abgeleiteten Klasse stehen

Zusammenfassen von catch-Blöcken

- Falls auf unterschiedliche Exceptions gleichartig reagiert werden soll, entstehen catch-Blöcke mit identischem Code
- Duplizierung von Code ist zu vermeiden (DRY-Prinzip)
- Ab Java Version 7 können catch-Blöcke zusammengefasst werden

```
//  
int[] operanden = new int[2];  
System.out.print("Zahl 1:");  
try{  
    operanden[i++] = sc.nextInt();  
} catch (ArrayIndexOutOfBoundsException |  
        NumberFormatException e){  
    //  
    e.printStackTrace();  
}  
//
```

fängt beide Ausnahmetypen



Java IO

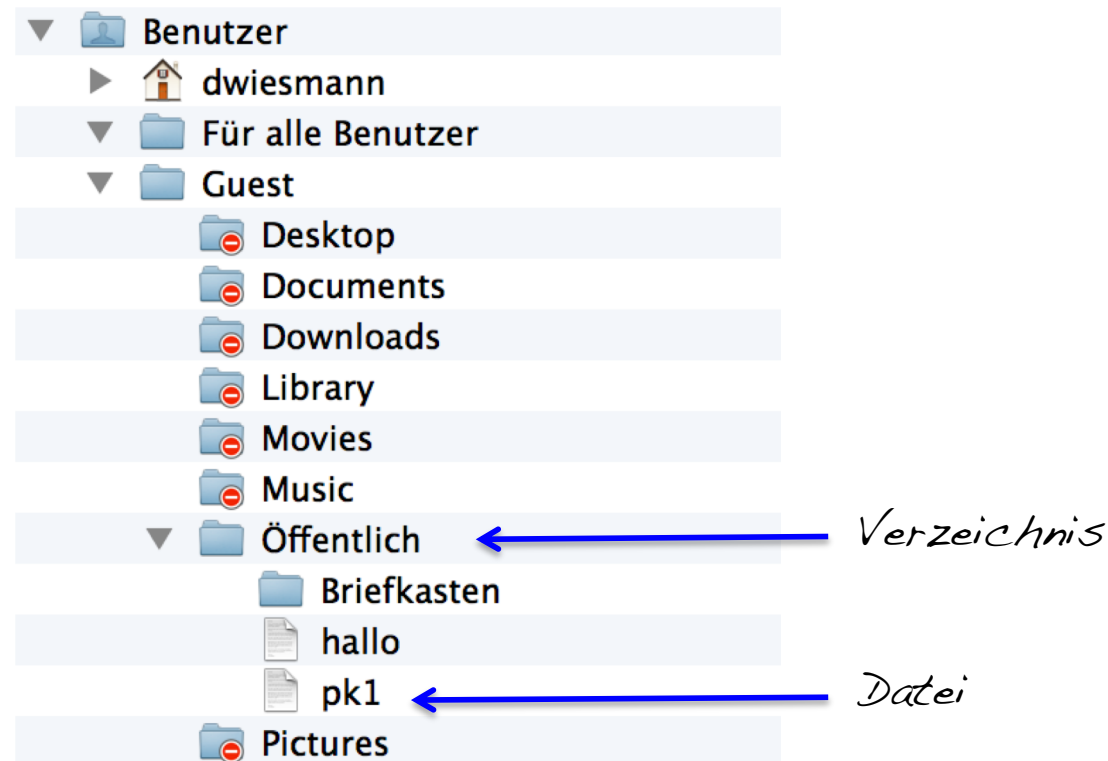
Dateien

- Objekte werden zur Laufzeit erzeugt und werden im Hauptspeicher gespeichert (Heap-Bereich)
 - Vorteil:
 - Direkte Adressierung
 - Schneller Zugriff auf den Objektzustand (die Attribute)
 - Nachteil:
 - Größe des zur Verfügung stehenden Hauptspeicher ist beschränkt (GByte-Bereich)
 - Der Zustand überdauert nicht das Prozessende
- Wir benötigen also eine dauerhafte (persistente) Speicherung von Daten

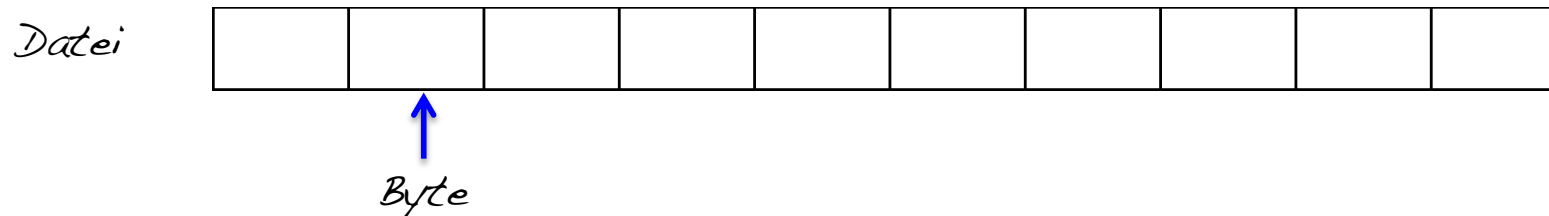
- Für die dauerhafte Speicherung von Daten werden sogenannte Massenspeicher als Datenträger eingesetzt
 - Magnetbänder
 - Festplatten
 - optische Speichermedien
- Die persistente Speicherung von Daten bietet die folgenden Vorteile
 - Daten überdauern das Prozessende
 - Große Datenmengen (TByte-Bereich)
 - Mehrere Prozesse können auf die Daten zugreifen
- Die physische Speicherung der Daten hängt vom verwendeten Datenträgertyp ab
- Als Anwendungsprogrammierer werden wir aber nicht mit den physikalischen Details der Datenspeicherung konfrontiert
 - Das Betriebssystem bietet eine geeignete Abstraktionsschicht

- Das Betriebssystem
 - fasst Daten zu Dateien zusammen
 - ermöglicht Zugriff über einen Dateinamen
 - ordnet Dateien Attribute zu
 - organisiert Dateien in Verzeichnissen
- Aufgabe 20: Nennen Sie fünf typische Dateiattribute!

- In der Praxis sind tausende von Dateien zu verwalten
 - Um die Dateien zu organisieren, können Dateien in Verzeichnisse gruppiert werden
 - Verzeichnisse sind hierarchisch angeordnet (in der Regel in einer Baumstruktur)



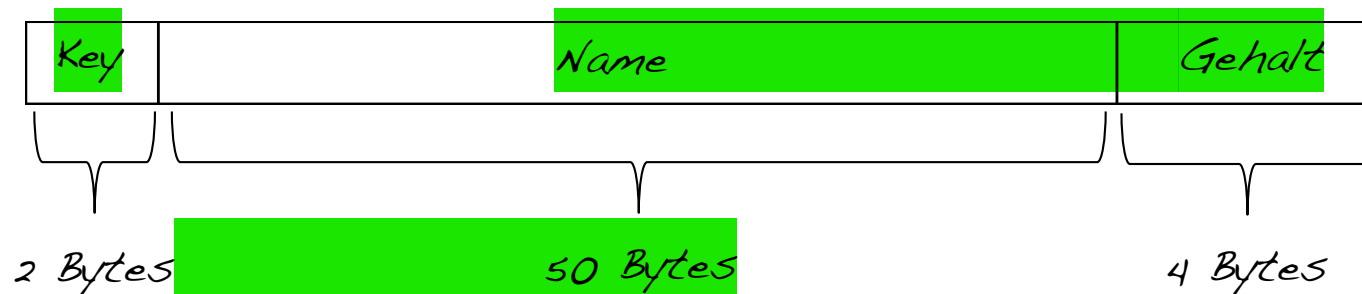
- Eine Datei ist zunächst eine unstrukturierte Folge von Bytes
 - die einzelnen Speicherzellen (Byte) haben keine Adresse
 - die Speicherzellen müssen vom Dateianfang an durchgezählt werden



- Aus fachlicher Sicht möchten wir aber keine Bytefolgen speichern, sondern z.B. Angestellte
 - Um eine logische Strukturierung zu erhalten, werden Dateien in Datensätze unterteilt
 - Jeder Datensatz hat eine feste Länge



- Jeder Datensatz hat eine interne Struktur (mit fester Länge)
 - Beispiel: Datensatz mit einer Länge von 56 Bytes zur Speicherung eines Angestellten



- Diese Form der linearen Speicherung von Datensätzen wird als sequentielle Speicherung bezeichnet

File

- Die Klasse `java.io.File` dient der Repräsentation von Datei- und Verzeichnisnamen
- Darüber hinaus bietet die Klasse `File` Möglichkeiten, um
 - auf Dateiattribute zuzugreifen
 - Dateien und Verzeichnisse anzulegen und zu löschen
- Ziel der Klasse `File`: Plattformunabhängigkeit

Trennen der Pfadbestandteile

Unix	/
Microsoft	\

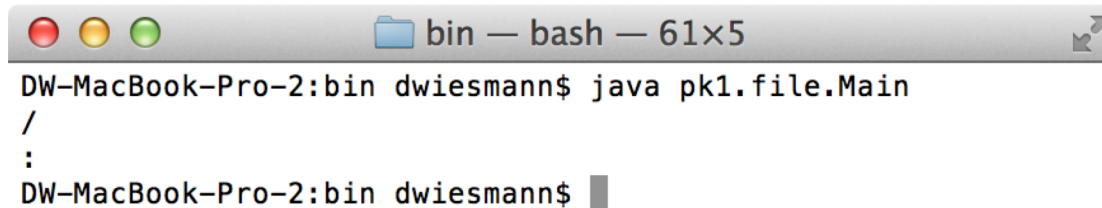
Trennen von Pfaden

Unix	:
Microsoft	;

– Beispiel

```
import java.io.File;

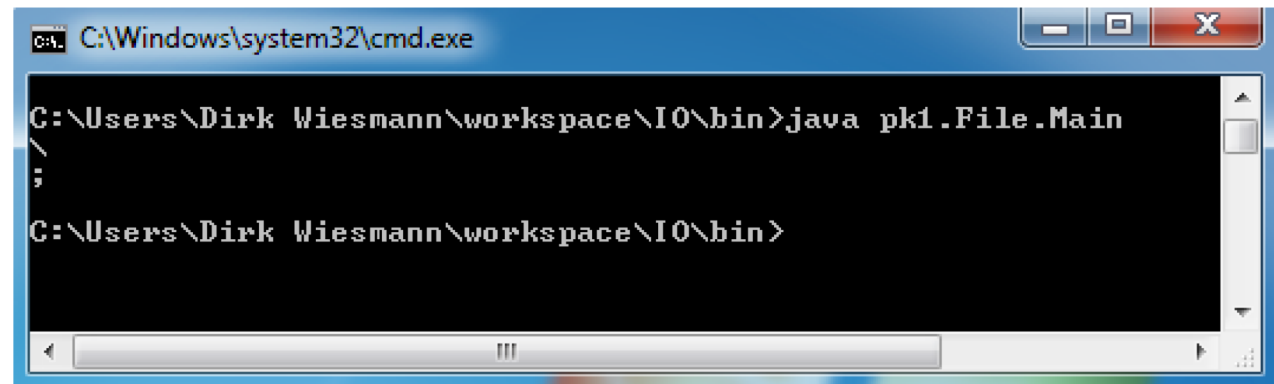
public class Main {
    public static void main(String[] args) {
        System.out.println(File.separator);
        System.out.println(File.pathSeparator);
    }
}
```

A screenshot of a macOS terminal window titled 'bin — bash — 61x5'. The prompt is 'DW-MacBook-Pro-2:bin dwiesmann\$'. The command 'java pk1.file.Main' has been executed, resulting in two lines of output: a forward slash '/' followed by a vertical colon ':'.

```
bin — bash — 61x5
DW-MacBook-Pro-2:bin dwiesmann$ java pk1.file.Main
/
:
DW-MacBook-Pro-2:bin dwiesmann$
```

OS X

Windows

A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The prompt is 'C:\Users\Dirk Wiesmann\workspace\IO\bin>'. The command 'java pk1.File.Main' has been executed, resulting in two lines of output: a forward slash '/' followed by a vertical colon ':'.

```
C:\Windows\system32\cmd.exe
C:\Users\Dirk Wiesmann\workspace\IO\bin>java pk1.File.Main
/
:
C:\Users\Dirk Wiesmann\workspace\IO\bin>
```

Ausgaben
sind unterschiedlich

- File kann absolute und relative Dateipfade verwalten
 - auf die Bestandteile des Pfades kann mit Methoden zugegriffen werden
- siehe API*

– Beispiel:

```
File f1 = new File("/Users/dwiesmann/Pk1/IO");  
File f2 = new File("./..");  
System.out.println(f1.getName());  
System.out.println(f1.getPath());  
System.out.println(f2.getAbsolutePath());  
try {  
    System.out.println(f2.getCanonicalPath());  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

← absoluter Pfad

← relativer Pfad

→ IO

→ Users/dwiesmann/Pk1/IO

→ Users/dwiesmann/Workspaces/Eclipse/VorlesungPK1/IO/../../

→ Dateizugriff möglich

→ Users/dwiesmann/Workspaces/Eclipse/VorlesungPK1

- Achtung: **File**-Objekte können mit ungültigen Dateinamen instanziiert werden (es findet kein Zugriff auf das Dateisystem statt)

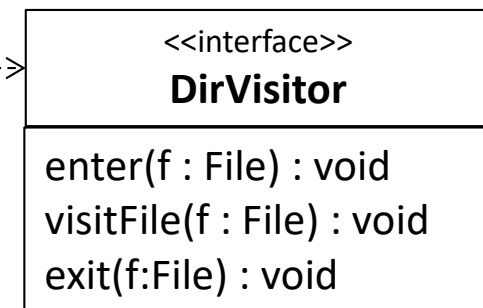
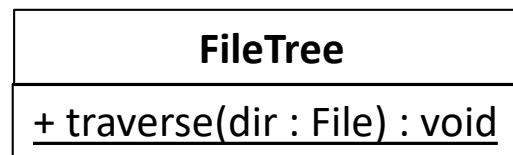
- Weitere Methoden der Klasse `File`

<code>String getParent()</code>	Gibt den Pfadnamen zum aktuellen Verzeichnis / zur aktuellen Datei an
<code>boolean isAbsolute()</code>	liefert genau dann <code>true</code> , wenn der Pfad absolut ist
<code>boolean isFile()</code>	liefert genau dann <code>true</code> , wenn der Pfadname eine Datei bezeichnet
<code>boolean isDirectory()</code>	liefert genau dann <code>true</code> , wenn der Pfadname ein Verzeichnis bezeichnet
<code>boolean exists()</code>	liefert genau dann <code>true</code> , wenn <code>isFile()</code> oder <code>isDirectory()</code> den Wert <code>true</code> liefern
<code>File[] listFiles()</code>	liefert ein Array mit den abstrakten Pfadnamen, die im Verzeichnis liegen

- Aufgabe 26: Schreiben Sie ein Programm, das einen kompletten Verzeichnisbaum durchläuft und alle Verzeichnisse und Dateien auf dem Bildschirm ausgibt. Dabei sollte die hierarchische Verzeichnisstruktur durch Einrückungen veranschaulicht werden

Beispiel

```
index.htm
microsoft.htm
netscape.htm
[diverses] ← Verzeichnis
[anzeige]
  farbnamen_16.htm
  farbnamen_netscape.htm
  farbpalette_16.htm
  farbpalette_216.htm
clients.htm
farbpaletten.htm
index.htm
mimetypen.htm
robots.htm
sprachenlaenderkuerzel.htm
[editorial]
  arbeitshinweise.htm
```



Idee: Der Durchlauf durch das Dateisystem verläuft immer gleich. Das Verhalten (z.B. Drucken, Dateien zählen, Speicherplatz berechnen) kann einfach über ein Interface ausgetauscht werden

*Io default test
NotBeans. ✓*