



# Java Erweiterungen I

Effiziente Java-Programmierung



Cegos Group

inspire  
qualify  
change



# Interaktive Dokumentation

- Sie können mit diesen Buttons ab der Seite „Inhaltsverzeichnis“ (nachfolgende Seite) **navigieren**.



	<i>Vorherige Seite, nächste Seite</i>		<i>Anfang des Kapitels</i>
	<i>Zurück zum Inhaltsverzeichnis</i>		<i>Link zu einer anderen Seite</i>



# Inhalts- verzeichnis



Einführung



Multithreading



Klassen und Interfaces



Java und Datenbanken



Enums und Collections



Anhang



Ein-/Ausgabe und Properties



GUI-Programmierung



# Kapitel 01: Einführung



Grundbegriffe



Laufzeitumgebung



Speicherorganisation



Objektorientierte  
Programmierung (OOP)



UML (Unified Modeling  
Language)



Einige Klassen der Bibliothek



Datenkapselung



Pakete



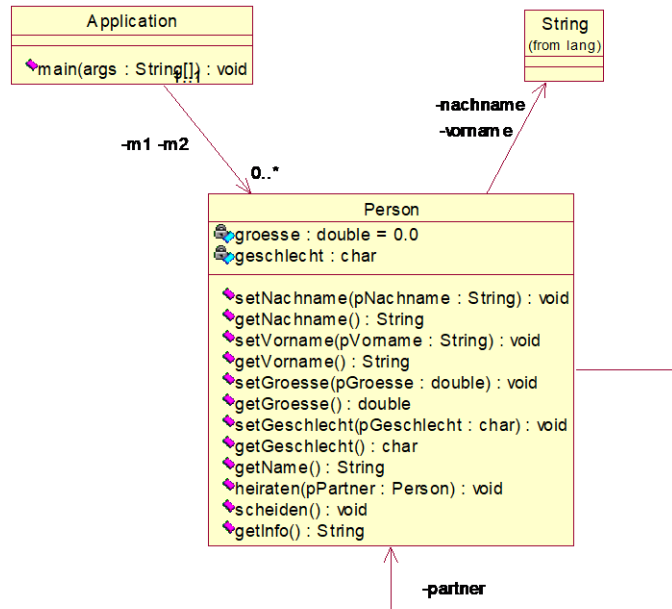
Variable Parameterliste



Schlüsselwörter und Komponenten  
der Java Standard Edition



# Grundbegriffe



- Grundlegende Kenntnisse der Java Entwicklungs-/Laufzeitumgebung
- Speicherorganisation
- Grundlagen objektorientierter Programmierung
  - Klassen, Objekte, Kapselung, Vererbung und Polymorphie
- Basis-Kenntnisse der Unified Modeling Language
  - Klassendiagramme
    - Klasse/Objekt
    - Attribut
    - Methode
    - Sichtbarkeit
    - Assoziation
    - Kardinalität
    - Vererbung/Generalisierung
- Benutzung der Runtime-Library (jre/lib/rt.jar)
  - Einige Klassen aus den packages: java.lang, java.util

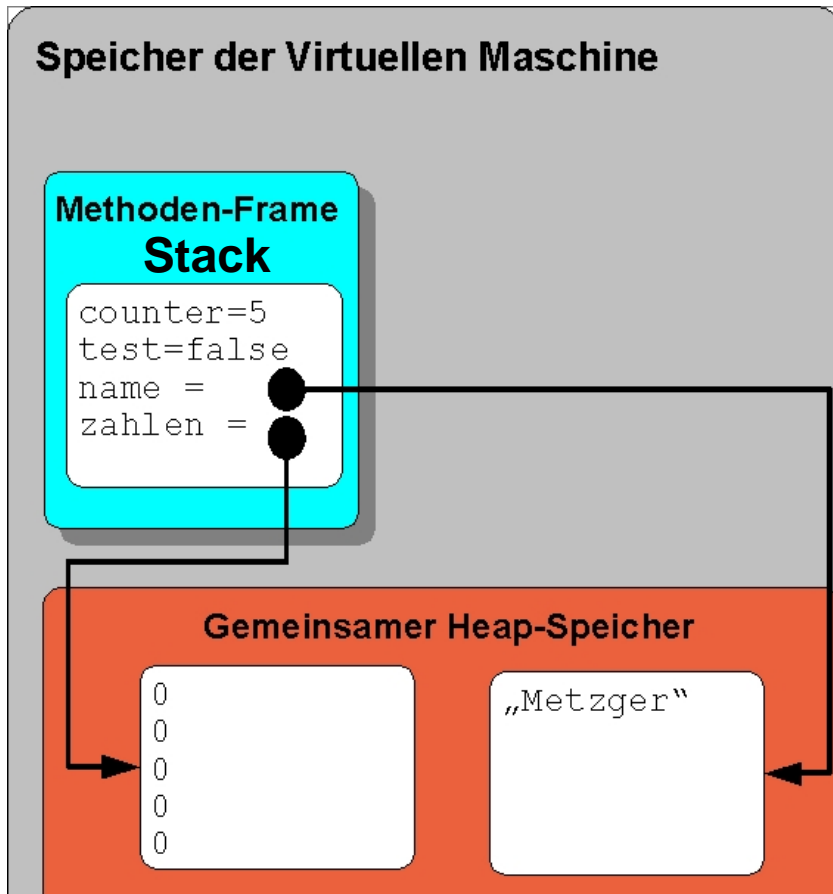


# Laufzeitumgebung

- Java ist eine typisierte objektorientierte Sprache
- Die Klassenbibliothek ist organisiert in Paketen:
  - voll qualifizierter Klassenname
  - Umgebungsvariable CLASSPATH
  - Datei rt.jar
- Java-Programme laufen plattform-unabhängig in einer virtuellen Maschine
  - Bytecode wird vom Java-Interpreter ausgeführt



# Speicherorganisation

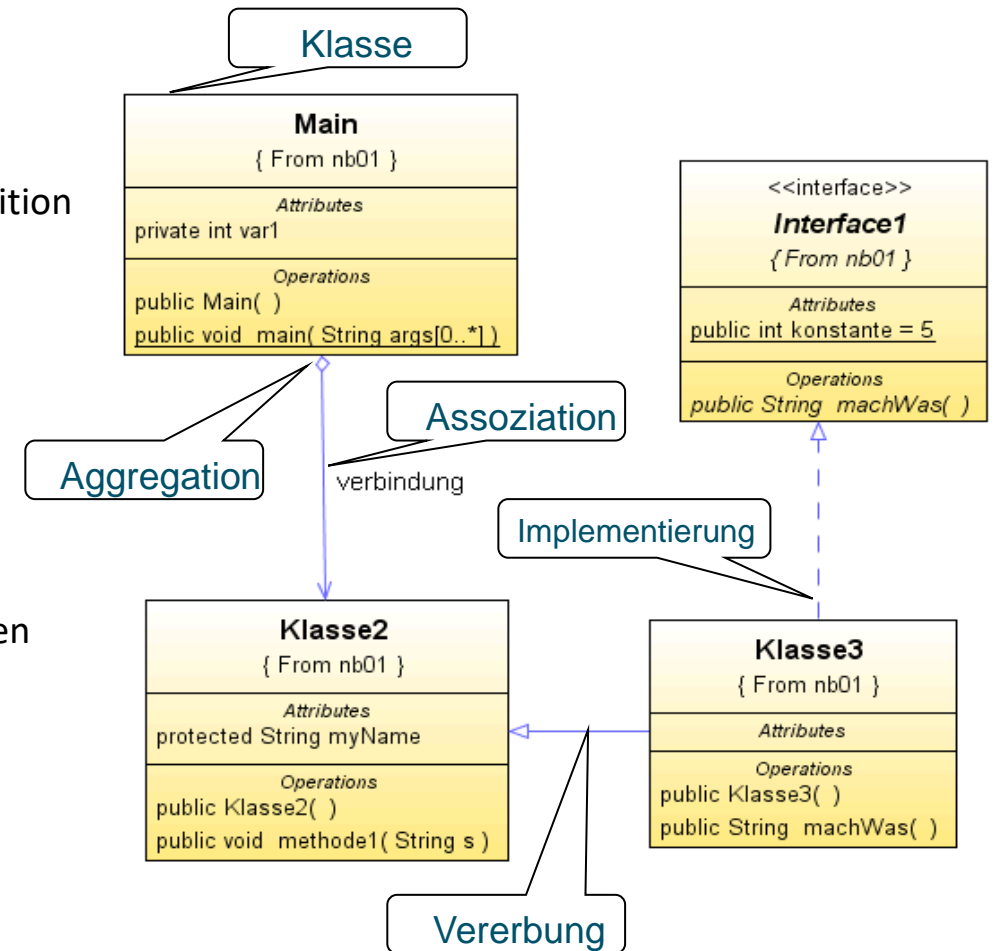


- Lokale Variable (Wert im Methoden-Frame)
  - Variable auf dem Stack:
    - haben nur eine begrenzte Lebensdauer
    - werden nicht automatisch initialisiert
- Objekt-Attribute
  - sind gültig, solange die Instanz existiert
  - werden automatisch initialisiert
- Einfache Variablen
  - werden durch einen Wert des entsprechenden Datentyps repräsentiert
- Referenzen und Arrays
  - Wert wird interpretiert als Adresse eines Bereiches im gemeinsamen Heap-Speicher
- Reservieren/Belegen von Speicher durch das Schlüsselwort **new**



# Objektorientierte Programmierung (OOP)

- OOP-Mechanismen:
  - Klassen, Objekte und Botschaften
  - Kapselung
  - Assoziation, Aggregation, Komposition
  - Konstruktoren
  - Einfach-Vererbung bei Klassen
  - Abstrakte Methoden und Klassen
  - Mehrfach-Vererbung bei Schnittstellen
  - Implementierung von Schnittstellen
- Polymorphie
- Definition von Signaturen in Interfaces
- Modellierung mittels UML-Klassendiagramm







# UML (Unified Modeling Language)



- **Modellierungssprache für objektorientierte Analyse**
- **Von der OMG (Object Modeling Group) Mitte der 90er entwickelt**
  - Bezog zahlreiche Vorschläge und gängige Vorgehensweisen ein
- **Definiert ein einheitliches Vokabular für das Erstellen von Modellen**
- **Sowohl statische als auch dynamische Aspekte können in der UML modelliert werden**
- **Vereinbart momentan 13 verschiedene Diagrammtypen (hier eine Auswahl)**
  - Paketdiagramm (statisch)
  - Klassendiagramm (statisch)
  - Kollaborationsdiagramm (dynamisch)
  - Sequenzdiagramm (dynamisch)



## Einige Klassen der Java Bibliothek

- String
- StringBuilder
- NumberFormat
- DateFormat
- Wrapperklassen
  - Byte
  - Short
  - Integer
  - Long
  - Float
  - Double
  - Boolean
  - Character



# Datenkapselung



- **private**

Zugriff nur innerhalb der deklarierenden Klasse möglich

- **<default>**

Zugriff nur innerhalb des Paketes möglich

- **protected**

Zugriff innerhalb der deklarierenden Klasse und  
innerhalb aller Subklassen und  
innerhalb des Paketes möglich

- **public**

Zugriff aus allen Klassen heraus möglich



# Pakete

- Leichtes Auffinden von Klassen
- Keine Namenskonflikte mit Klassen in anderen Paketen
- Festlegung von Beziehungen zwischen verwandten Klassen
- Eine zusätzliche Ebene der Kapselung
- Statische Importe seit Java 5
- Paketzuordnung:  
`package de.integrata.java.erweiterungen.oop;`
- Import:  
`import de.integrata.java.erweiterungen.oop.*;`  
`import de.integrata.java.erweiterungen.oop.Klasse;`  
im Code: `de.integrata.java.erweiterungen.oop.Klasse`



# Variable Parameterliste

- Übergabe beliebig vieler Parameter an eine Methode

```
static int sum(int... args) {  
    int sum = 0;  
    for (int arg : args)  
        sum += arg;  
    return sum;  
}
```

- Verwendung

```
System.out.println(sum(1, 2));  
System.out.println(sum(1, 2, 3));  
System.out.println(sum(1, 2, 3, 4));
```

Tatsächlich wird bei all diesen Aufrufen immer nur ein einziger Parameter übergeben: ein Array mit int-Werten.

- Einschränkung:

- Nur ein Parameter kann variabel sein
- Nur der letzte Parameter kann variabel sein



# Schlüsselwörter

abstract

continue

for

new

switch

assert

default

(goto)

package

synchronized

boolean

do

if

private

this

break

double

implements

protected

throw

byte

else

import

public

throws

case

enum

instanceof

return

transient

catch

extends

int

short

try

char

final

interface

static

void

class

finally

long

strictfp

volatile

(const)

float

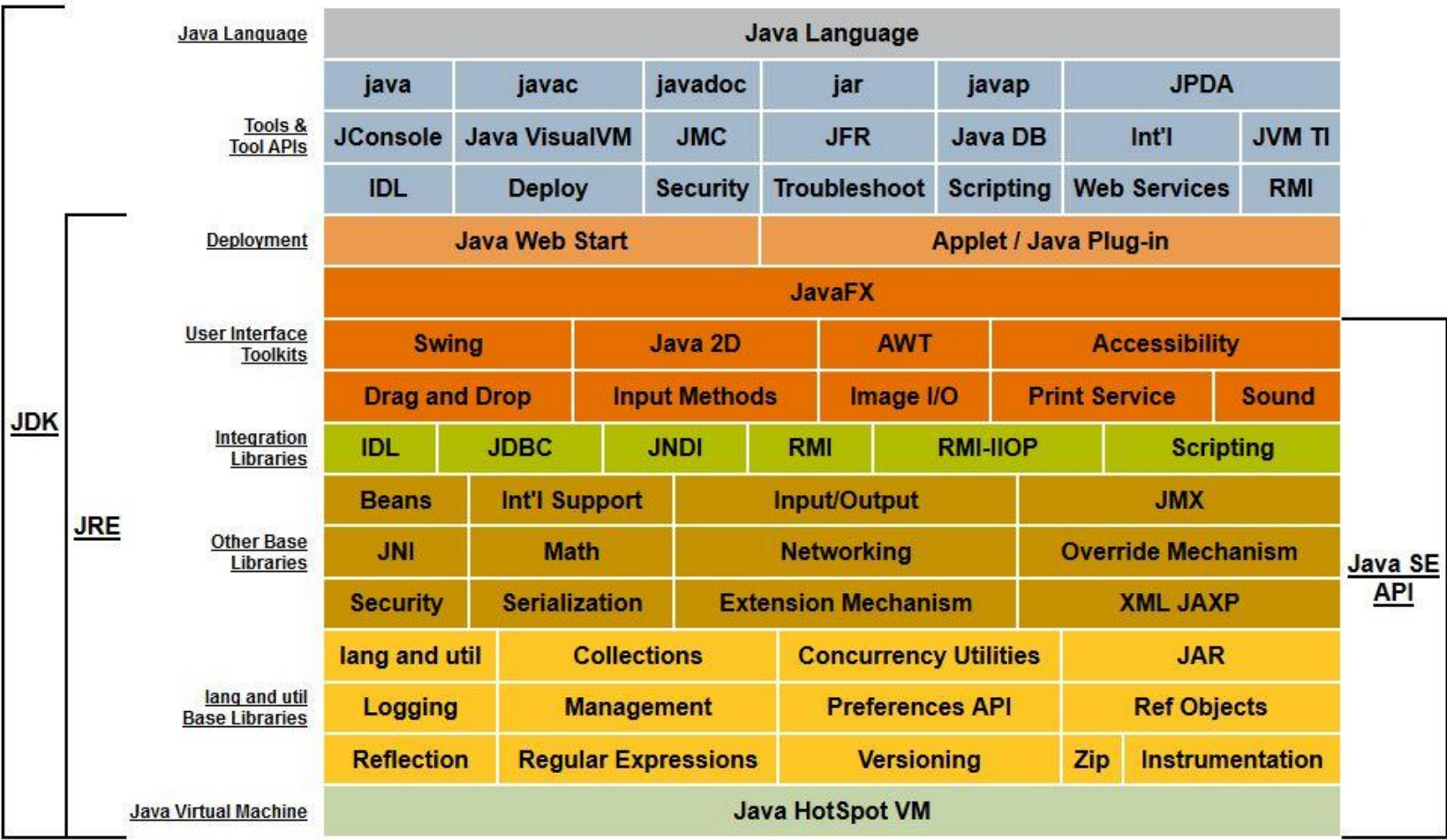
native

super

while



# Komponenten der Java Standard Edition



© <https://docs.oracle.com/javase/8/docs/>



# Kapitel 02: Programmierung



Klasse Object



Reflection



Garbage Collector



Ausnahmebehandlung



Typen von Klassen



Interface





# Klasse Object

## Java.lang.Object

```
+ Object( )
# clone( ): Object
+ equals(obj: Object): boolean
# finalize( )
+ getClass( ): Class
+ hashCode( ): int
+ notify( )
+ notifyAll( )
+ toString( ): String
+ wait( )
+ wait(timeout: long)
+ wait(timeout: long, nanos: int)
```

### Wichtige Methoden

#### String toString( )

- Zeichenkettendarstellung, Standard: Klassenname + @ + Hashcode, Modifikation durch Überschreiben.

#### boolean equals( Object )

- Vergleich von Objekten, Standard: Identität, Modifikation durch Überschreiben
- Zu einer Implementierung von equals( ) sollte immer eine Implementierung von hashCode( ) gehören, denn wenn zwei Objekte gleich sind, müssen auch die Hashwerte gleich sein.

#### int hashCode( )

- Die Methode hashCode() berechnet zu einem Objekt einen Hash-Code. Der Hash-Code Wert wird verwendet, um Objekte in einem hash-basierten Container zu finden oder dort abzulegen.

#### Object clone( )

- Kopie eines Objektes, Standard-Implementierung wirft die : CloneNotSupportedException
  - Aktivieren mit implements Cloneable
  - flache Kopie mit super.clone()
  - tiefe Kopie mit Überschreiben.



## Garbage Collector

- kettet unbenutzten Objekt-Speicher wieder in die Speicherkette.
- zerstört Objekte bzw. räumt den heap-Speicher auf.
- kann nicht direkt aufgerufen werden sondern läuft als unabhängiger Thread.
- kann angestoßen werden (triggern) durch **System.gc( )**.
- Bevor ein Object endgültig zerstört wird, wird die in Object definierte `finalize( )`-Methode (seit JDK 9 deprecated) aufgerufen.
  - `protected void finalize() throws Throwable`



# Ausnahme- behandlung



- Tritt während der Programmausführung ein Fehler auf, wird die normale Programmausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse Throwable fasst alle Arten von Fehlern zusammen.
- Ein Fehlerobjekt kann gefangen und geeignet behandelt werden.
- Trennung von normalem Ablauf und Behandlung von Sonderfällen (wie fehlerhaften Eingaben, ...)



# Klassen zur Ausnahme- behandlung



- **Throwable** (Superklasse von Error und Exception)
- **Error**
  - für fatale Fehler, die zur Beendigung des gesamten Programms führen.
- **Exception**
  - für behandelbare Fehler oder Ausnahmen.
- **RuntimeException** (Subklasse von Exception)
  - fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen. Sie kann jederzeit auftreten und kann, muss aber nicht abgefangen werden.
- **unchecked exception**
  - Ausnahmen der Klasse Error und RuntimeException müssen nicht im Methodenkopf deklariert werden.
- **checked exception**
  - Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode deklariert werden.



# Schlüsselwörter

- **try**
  - Definiert einen Block, innerhalb dessen Ausnahmen (Exceptions) auftreten können (geworfen werden können).
- **catch**
  - Definiert einen Block, der die Fehlerbehandlung für die durch den im catch-Befehl angegebenen Ausnahmetyp durchführt.
- **finally**
  - Definiert einen Block, der stets ausgeführt wird, egal ob ein Fehler auftrat oder nicht. Auch wenn
    - innerhalb des catch-Zweiges eine weitere Exception geworfen wird.
    - im catch-Zweig ein return steht.
    - kein catch-Zweig durchlaufen wird.
- **throw**
  - Erzeugt (wirft) eine Ausnahme. Hierfür muss dem Befehl throw ein Objekt übergeben werden, das eine Unterklasse von Throwable ist (dies sind die Klassen Exception und Error).
- **throws Exception**
  - Die Methode muss mit throws eine Liste aller Ausnahmen definieren, die geworfen werden können.



# Klassentypen

- Globale Klasse
  - Organisation in Paketen
  - Bytecode: Klasse.class
- Innere Klasse (Memberklasse)
  - Klasse in Klasse
  - Bytecode: KlasseAussen\$KlasseInnen.class
- Statische innere Klasse
  - Static Klasse in Klasse kann ohne Instanz der äußeren Klasse Instanziiert werden
  - Bytecode: KlasseAussen\$KlasseStaticInnen
- Lokale Klasse
  - Klasse in Methode
  - Bytecode: KlasseAussen\$1KlasseLokal.class
- Anonyme Klasse
  - Namenlose Klasse in Methode
  - Bytecode: KlasseAussen\$1.class



# Beispiel

## Globale Klasse

```
public class Klasse {
    // Konstruktor Klasse
    public Klasse ( ) {
        System.out.println( "Klasse erzeugt" );
    }
} // class KlasseAussen
```

### ■ Instanzerzeugung

```
Klasse k = new Klasse ( );
```

Ausgabe:

Klasse erzeugt





# Beispiel

## Innere Klasse

Ausgabe:

KlasseAussen erzeugt  
KlasseInnen erzeugt

```
public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );
    }

    public class KlasseInnen {
        // Konstruktor Innere Klasse
        public KlasseInnen( ) {
            System.out.println( "KlasseInnen erzeugt" );
        }
    } // class KlasseInnen
} // class KlasseAussen
```

### ■ Instanzerzeugung

```
KlasseAussen ka = new KlasseAussen( );
KlasseAussen.KlasseInnen ki = ka.new KlasseInnen( );
```





# Beispiel Statische Innere Klasse

Ausgabe:

KlasseStaticInnen erzeugt

```
public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );
    }

    public static class KlasseStaticInnen {
        // Konstruktor Innere static Klasse
        public KlasseStaticInnen( ) {
            System.out.println( "KlasseStaticInnen erzeugt" );
        }
    } // class KlasseStaticInnen
} // class KlasseAussen
```

## ■ Instanzerzeugung

```
KlasseAussen.KlasseStaticInnen ki = new KlasseAussen.KlasseStaticInnen( );
```



# Beispiel Lokale Klasse

Ausgabe:

KlasseAussen erzeugt  
KlasseLokal erzeugt

```
public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );
    }

    public void methodeAussen( ) {
        class KlasseLokal{
            // Konstruktor Lokale Klasse
            KlasseLokal( ){
                System.out.println( "KlasseLokal erzeugt" );
            }
        } // KlasseLokal
        new KlasseLokal( );
    } // MethodeAussen
} // class KlasseAussen
```

## ■ Instanzerzeugung

```
KlasseAussen ka = new KlasseAussen( );
ka.methodeAussen( );
```



# Beispiel

## Anonyme Klasse

## Ausgabe:

KlasseAussen erzeugt  
Implementierung Anonym

```
public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );

        // Anonyme Klasse
        doSomething( new Object( ) {
            public String toString( ) {
                return "Implementierung Anonym";
            }
        } );
    }

    private void doSomething( Object pObject ) {
        System.out.println( pObject.toString( ) );
    } // doSomething
} // class KlasseAussen
```

## ■ Instanzerzeugung

```
KlasseAussen ka = new KlasseAussen( );
```



# Ein Interface (Schnittstelle) kann enthalten



- Abstrakte Methoden
- Implementierte (default) Methoden seit Java 8
- Statische Konstanten
- Statische Methoden
- Private Methoden seit Java 9
- Private statische Methoden seit Java 9

```
public interface Foo {
    int x = 42; //implizit public static final
    final int y = 43; //implizit public static
    public static final int z = 44;

    void f(); //implizit public abstract
    public void g(); //implizit abstract
    public abstract void h();

    class C { }; //implizit public static
    public static class D { }
}
```



# Die Klasse `class`

- Instanzen der Klasse `Class` repräsentieren Klassen und Interfaces in einer laufenden Java-Anwendung.
  - Ein Speicherbereich auf dem Heap, der dem Typ zugeordnet ist.
  - engl. oft "**class object**" genannt
- Einstiegspunkt für die Introspektion (= Erkenntnisse über die eigene Struktur gewinnen)
  - `<Klassenname>.class`
  - `<objectReferenz>.getClass( )`
- Einige Methoden:
  - `static Class.forName( String className ) // Klasse laden`
  - `java.lang.reflect.Constructor[ ] getConstructors( )`
  - `java.lang.reflect.Field[ ] getFields( )`
  - `java.lang.reflect.Method[ ] getMethods( )`
  - `Object newInstance( )`



# Dynamische Instanzerzeugung

- Was bedeutet dynamisch?
  - Zur Laufzeit soll ein Objekt erzeugt werden, das zur Compilezeit noch nicht fest definiert wird.
- Aufgabe
  - Erzeuge ein Objekt einer Klasse, dessen Typ erst zur Laufzeit ermittelt wird.
- Lösung
  - Der Typ des zu erstellenden Objektes soll als String angegeben werden.
  - Der String kann zur Laufzeit flexibel ermittelt werden.
    - GUI-Eingabe, Datei, Netzwerk, ...
  - nötige Syntax **new "Klassenname" () ;** existiert nicht.



# Kapitel 03: Enums und Collections



Collections



List



Set



Map



Generische Collections



Collections und einfache Datentypen



Hilfsklassen



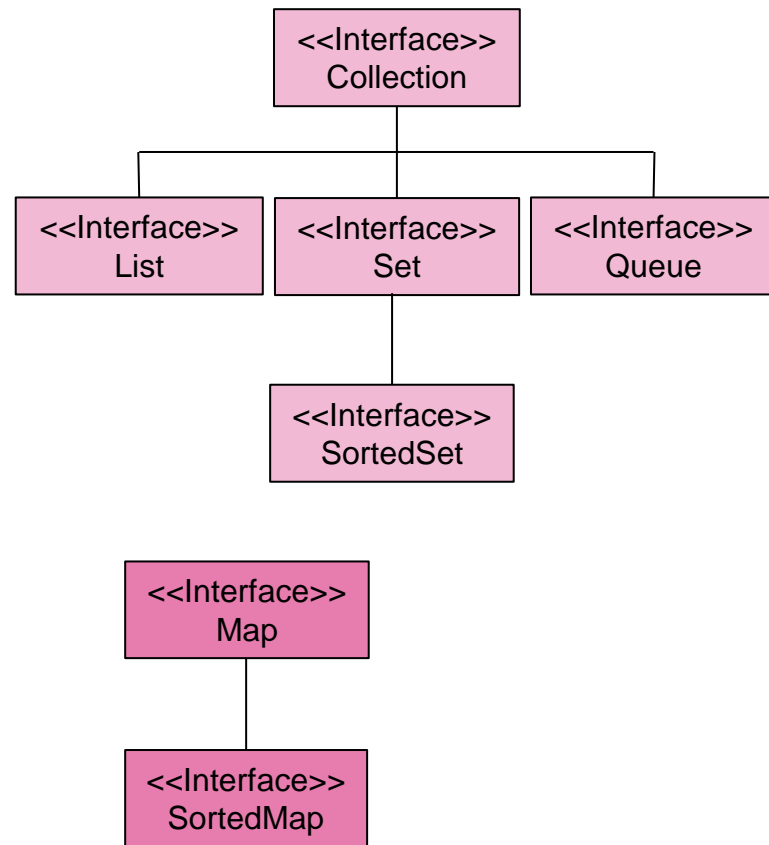
Sortieren von Elementen



Enums



# Collection-Framework



## ■ List

- eine dynamische Liste

## ■ Set

- eine Liste, die jedoch keine doppelten Einträge enthalten darf

## ■ Map

- bildet Elemente eines Typs auf Elemente eines anderen Typs ab

## ■ Queue (seit Java 5)

- eine dynamische Collection mit speziellen Methoden zum Einfügen und Entfernen von Elementen

## ■ Iterator

- Zugriff auf die Elemente der Collection in einer Schleife
- Methoden: **hasnext**, **next** **remove**

## ■ ListIterator

- Zusätzlich zum **Iterator** ist auch das Hinzufügen und Löschen von Elementen sowie ein Rückwärtsblättern vorgesehen





# Implementierungen von List

- **ArrayList**
  - Intern wird die Liste als Array gehalten
  - Schneller Zugriff über Index
  - Relativ langsam beim Einfügen innerhalb der Liste
- **LinkedList**
  - Verwaltung durch eine doppelt verkettete Liste
  - Relativ langsam beim Zugriff über den Index
  - Sehr schnell beim Einfügen und Entfernen
- **Vector**
  - Wie die **ArrayList**, jedoch vor gleichzeitigen Zugriffen geschützt (thread-safe)



# Implementierungen von Set

- **HashSet**
  - Schnelle Set-Implementierung (mit HashMap), die keine Aussagen über die Reihenfolge der Elemente bei der Iteration macht.
- **LinkedHashSet**
  - Die Reihenfolge der Elemente bei der Iteration entspricht garantiert der Reihenfolge beim Hinzufügen
- **TreeSet**
  - Implementiert die Schnittstelle **SortedSet**, die Suchfunktionalitäten definiert
- **EnumSet**
  - **Set**, das nur Elemente eines einzigen `enum`-Typs aufnehmen kann



# Implementierungen von Map

- **HashMap**
  - Map, bei der die Iteration über die Elemente nicht der Reihenfolge des Hinzufügens entspricht
- **LinkedHashMap**
  - Die Reihenfolge der Elemente bei der Iteration entspricht garantiert der Reihenfolge beim Hinzufügen
- **TreeMap**
  - Implementiert die Schnittstelle **SortedMap**, die Suchfunktionalitäten definiert
- **EnumMap**
  - **Map**, die nur Schlüssel eines einzigen **enum**-Typs akzeptiert



# Generische Collections

- Generische Datentypen erlauben eine typsichere Verwendung der Collection-Klassen
  - Bei der Deklaration der Collection wird bereits der Typ der verwendbaren Elemente bestimmt.
- Verwendung spezieller Platzhalter-Klassen bei der Deklaration und Erzeugung der Collection

- Beispiel ArrayList

```
List<String> liste = new ArrayList<String>( );  
liste.add("Test");
```

- Beispiel Map

```
Map<Integer, String> table = new HashMap<Integer, String>( );  
Integer tableKey = new Integer(5);  
String tableValue = "Test";  
table.put(tableKey, tableValue);  
String right = table.get(tableKey);
```



# Collections und einfache Datentypen

- Autoboxing/Unboxing
  - automatisch Wandlung von einfachen Datentypen (`int`, `long`, ...) in die entsprechende Wrapper-Klasse und zurück.
- Dadurch ist die Benutzung von Collections mit einfachen Datentypen möglich
  - `Map<Integer, Double> intDoubleMap = new HashMap<Integer, Double>();`
  - `intDoubleMap.put(42, 18.60);`



# Hilfsklassen

## ■ Collections

- **Achtung:** nicht verwechseln mit dem Interface **Collection**)
- Hilfsklasse für **List**, **Set** und **Map**
- enthält allgemein verwendbare static Methoden
  - `copy`, `fill`, `list`, `max`, `min`, `reverse`, `shuffle`, `sort`, `swap`,  
...

## ■ Arrays

- Hilfsklasse für Arrays
- enthält allgemein verwendbare static Methoden zur Verwendung bei Arrays
  - `binarySearch`, `equals`, `fill`, `hashCode`, `sort`, `toString`, ...



# Sortieren von Elementen

## ■ Interface `java.lang.Comparable`

- Ermöglicht eine natürliche Folge der implementierenden Objekte.
- Lists (und arrays), die dieses Interface implementieren, können `Collections.sort` (and `Arrays.sort`) sortiert werden.
- Objekte können als Schlüssel/Elemente in sortierten Maps/Sets verwendet werden.
- `public int compareTo(Object pToCompare) ;`
  - Rückgabewert negativ, 0, positiv
  - für natürliche Reihenfolge der Objekte, kompatibel mit equals()

## ■ Interface `java.util.Comparator`

- Kann alternativ benutzt zum Sortieren werden (Übergabe an `Collections.sort`).
- `public int compare(Object pObj1, Object pObj2) ;`
  - Rückgabewert negativ, 0, positiv
  - für andere Sortierreihenfolge in einer Anwendung



# Sortieren von Elementen (Beispiel)

- Beispiel: Erzeugung einer List mit 3 Einträgen vom Typ String

```
List<String> tage = Arrays.asList( "Montag", "Dienstag", "Sonntag" );
```

- Sortierung der Elemente aufsteigend nach Länge des Strings mit einer anonymen Klasse

```
Collections.sort( tage,  
    new Comparator<String>( ) {  
        public int compare(String s1, String s2) {  
            return s1.length() - s2.length();  
        }  
    } );
```

- Sortierung der Elemente mit Lambda-Ausdruck

```
Collections.sort( tage,  
    ( String s1, String s2) -> { return s1.length() - s2.length(); } );
```

- Einfacher

```
Collections.sort( tage,  
    (String s1, String s2) -> s1.length() - s2.length() );
```

- Noch einfacher

```
Collections.sort( tage, (s1, s2) -> s1.length() - s2.length() );
```





# Enums:

## Ein Problem

- Verwaltung von vier Jahreszeiten

```
public interface Season {  
    public static final int SPRING = 0;  
    public static final int SUMMER = 1;  
    public static final int AUTUMN = 2;  
    public static final int WINTER = 3;  
}
```

- Aufruf

```
printSeason(Season.SPRING) ;  
printSeason(Season.SUMMER * 2) ;  
printSeason(42) ;
```



# Enums: Lösung mit „altem“ Java

## ■ Verwaltung von vier Jahreszeiten

```
public class Season {  
    public static final Season SPRING = new Season();  
    public static final Season SUMMER = new Season();  
    public static final Season AUTUMN = new Season();  
    public static final Season WINTER = new Season();  
  
    private Season() {  
    }  
}
```

## ■ Aufruf

```
static void printSeason(Season season) {  
    if (season == Season.SPRING)  
        System.out.println("Fruehling");  
    else if (season == Season.SUMMER)  
        System.out.println("Sommer");  
    else if (season == Season.AUTUMN)  
        System.out.println("Herbst");  
    else if (season == Season.WINTER)  
        System.out.println("Winter");  
}
```



# Enums: Lösung mit „altem“ Java etwas intelligenter

## ■ Verwaltung von vier Jahreszeiten

```
public class Season {  
    public static final Season SPRING = new Season(0, "SPRING");  
    public static final Season SUMMER = new Season(1, "SUMMER");  
    public static final Season AUTUMN = new Season(2, "AUTUMN");  
    public static final Season WINTER = new Season(3, "WINTER");  
  
    private final int ordinal;  
    private final String name;  
  
    private Season(int ordinal, String name) {  
        this.ordinal = ordinal;  
        this.name = name;  
    }  
  
    public final int ordinal() {  
        return this.ordinal;  
    }  
  
    public final String name() {  
        return this.name;  
    }  
  
    @Override  
    public String toString() { return this.name; }  
}
```



# Enums: Lösung mit „altem“ Java noch intelligenter

## ■ Verwaltung von vier Jahreszeiten

```
public class Season {  
    private static Map<String, Season> seasons =  
        new HashMap<String, Season>();  
    public static final ...  
    private final ...  
    private Season(int ordinal, String name) {  
        this.ordinal = ordinal;  
        this.name = name;  
        seasons.put(name, this);  
    }  
    public static Season valueOf(String name) {  
        Season s = seasons.get(name);  
        if (s == null) throw new IllegalArgumentException( );  
        return s;  
    }  
    public static Season[] values() {  
        return new Season[] { SPRING, SUMMER, AUTUMN, WINTER };  
    }  
    public final int ordinal() { return this.ordinal; }  
    public final String name() { return this.name; }  
    public String toString( ) { return this.name; }  
}
```



# Enums:

## Lösung seit Java 5

- Verwaltung von vier Jahreszeiten

```
public enum Season {  
    SPRING, SUMMER, AUTUMN, WINTER  
}
```

- Aufruf

```
static void printSeason2(Season season) {  
    switch(season) {  
        case SPRING:  
            System.out.println("Fruehling"); break;  
        case SUMMER:  
            System.out.println("Sommer"); break;  
        case AUTUMN:  
            System.out.println("Herbst"); break;  
        case WINTER:  
            System.out.println("Winter"); break;  
    }  
}
```



# Kapitel 04: Ein-/Ausgabe und Properties



Einführung in Java I/O



Klasse Properties



Klasse File



Klasse Preferences



Byteweise Lesen und Schreiben



Zeichenweise Lesen und Schreiben



Zeilenweise Lesen und Schreiben



# Einführung in Java I/O

- Paket `java.io`
- Funktionalität:
  - Verwaltung von Verzeichnis-/Datei-Pfaden
    - **File**
  - Lesen und Schreiben von Dateien
    - **Reader, Writer,**
      - für 16-bit Text-I/O
    - **InputStream, OutputStream**
      - für 8-bit Binär-I/O
  - Austausch von Objekten im gemeinsamen Speicher
  - Austausch von Objekten zwischen verschiedenen virtuellen Maschinen
- Hier: Nur Lesen und Schreiben von Dateien



# Klasse File

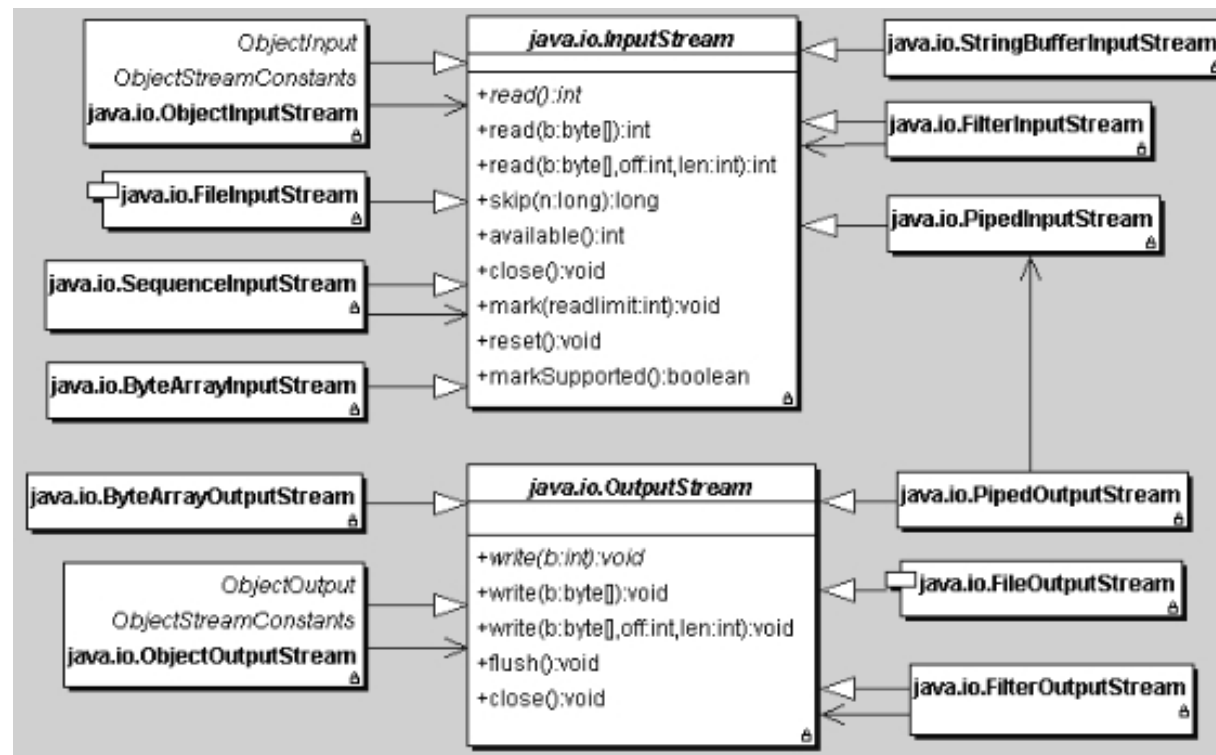
- Instanziierung unter Angabe eines Pfadnamens
  - Relative und absolute Angaben sind möglich
- Trennzeichen variieren zwischen den verschiedenen Betriebssystemen
  - ' \ ' unter Windows
  - ' / ' unter Unix/Linux
- Java Laufzeitumgebung definiert Trennzeichen
  - Systemvariable `file.separator`
  - Auslesen mit: `System.getProperty("file.separator")`
  - Klassenattribut `File.separator`





# Byteweise Lesen und Schreiben

- Lesen und schreiben durch Verwendung der Stream-Klassen
- InputStream und OutputStream**
  - Byte-basierte Datenströme





## Beispiel:

### try

### Schreiben einer

### Byte-Datei

```
OutputStream out = null;
try {
    out = new FileOutputStream("demo.bin");
    out.write( 65 );
}
catch (IOException e) {
    System.out.println(e);
}
finally {
    try {
        if (out != null)
            out.close();
    }
    catch (IOException e) {
        System.out.println(e);
    }
}
```



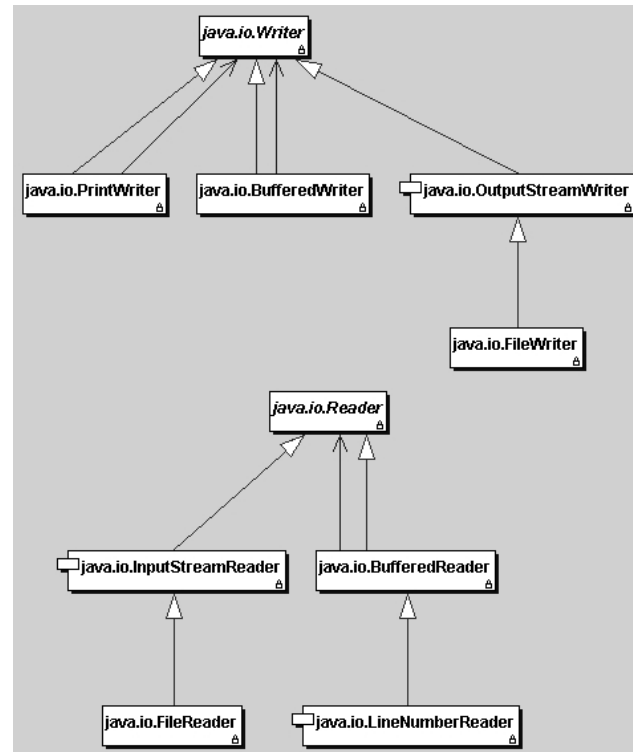
## Beispiel: try mit Ressourcen Schreiben einer Datei

```
try ( final OutputStream out = new
      FileOutputStream("demo.bin")
    ) {
    out.write( 65 );
}
catch (IOException e) {
    System.out.println(e);
}
```



# Zeichenweise Lesen und Schreiben

- Lesen und schreiben durch Verwendung der Stream-Klassen
- **Reader** und **Writer**
  - **char**-basierte Datenströme





## Beispiel: Schreiben einer Zeichen-Datei

```
try (final Writer writer = new
    FileWriter("demo.txt") ) {
    writer.write( "Eins" );
}
catch (IOException e) {
    System.out.println(e);
}
```



## Beispiel: Gepuffertes Schreiben einer Zeichen-Datei

```
try ( BufferedWriter writer = new  
        BufferedWriter( new FileWriter(  
                            "datei.txt" ) ) ) {  
    writer.write("Eins");  
    writer.newLine();  
    writer.write("Zwei");  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```



# Klasse Properties

	java.util.Hashtable
<b>java.util.Properties</b>	
<div>+Properties() +Properties(java.util.Properties) +setProperty(java.lang.String,java.lang.String):java.lang.Object +load(java.io.InputStream):void +save(java.io.OutputStream,java.lang.String):void +store(java.io.OutputStream,java.lang.String):void +getProperty(java.lang.String):java.lang.String +getProperty(java.lang.String,java.lang.String):java.lang.String +propertyNames():java.util.Enumeration +list(java.io.PrintStream):void +list(java.io.PrintWriter):void</div>	

- Erweiterung der `Hashtable`
- Properties lesen und schreiben
  - `String getProperty(String key)`
  - `void setProperty(String key, String value)`
- Lesen und speichern der Properties durch die Methoden
  - `load(InputStream pIn)`
  - `store (OutputStream pOut, String pHeader)`



# Klasse Preferences

- Benutzerabhängige Konfigurationseinstellungen
  - Properties-Dateien ablegen im User-Verzeichnis (System-Property: `user.home`)
  - oder Benutzung von `java.util.prefs.Preferences`
- Ablage der Preferences
  - bei UNIX
    - im User-Home Verzeichnis
  - unter Windows
    - in der Registry
- Hierarchische Organisation mit den Einstiegspunkten
  - `public static Preferences userRoot()`
  - `public static Preferences systemRoot()`
  - Setzen von Informationen durch `put`-Methoden





# Kapitel 05: GUI-Programmierung



Grafische Benutzeroberfläche



Abstract Window Toolkit (AWT)



Swing



LayoutManager



FlowLayout



BorderLayout



GridLayout



GridBagLayout



Eventhandling



Zusammenfassende Wertung



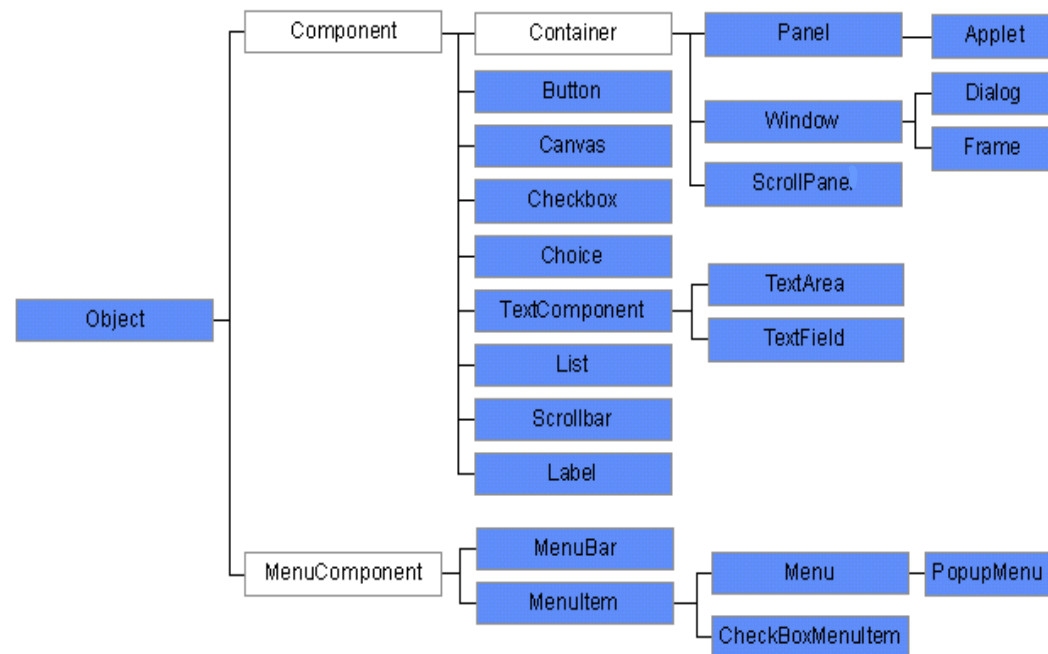
# Grafische Benutzeroberfläche

- GUIs auf allen Plattformen
  - AWT-Klassen sind die plattformunabhängige Schnittstelle für den Programmierer
  - Peer-Klassen sind plattformabhängige, „schwergewichtige“ Klassen, die Betriebssystem-Aufrufe durchführen
- Java Foundation Classes – Swing
  - Eigener Fenster-Manager mit plattformunabhängigen Komponenten
  - sog. „lightweight“-Klassen, einstellbares look&feel
- Standard Widget Toolkit
  - Eingeführt durch Eclipse
  - Peer-Klassen wie AWT aber wesentlich umfangreicher
  - <http://www.eclipse.org/swt/>
- Java FX
  - JavaFX-UI-Toolkit: FXML ([XML](#)-basierte Sprache) zum Erstellen von Oberflächen
  - Ein *scene graph* verwaltet die einzelnen Bestandteile einer GUI



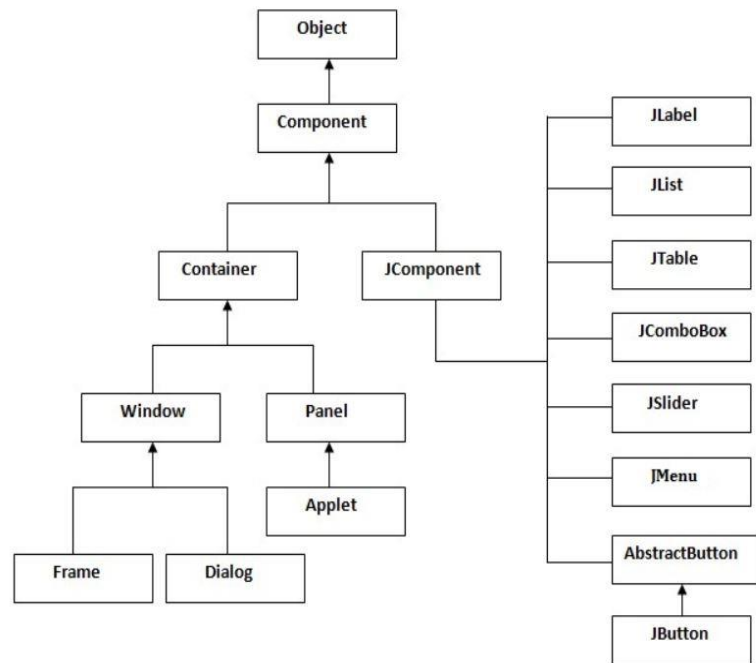
# Abstract Window Toolkit (AWT)

- Java AWT-Klassen können in drei Gruppen eingeteilt werden:
  - Komponenten** (Button, Menu, List u.s.w.)
  - Layout-Manager** (BorderLayout, FlowLayout, GridLayout u.s.w.)
  - Utility-Klassen** (Graphics, Font, Color, Polygon u.s.w.)





# Swing



- Pakete **javax.swing** und Unterpakete (Klassen beginnen mit **J** z.B. **JButton**)
- 100 % in Java geschrieben ("leichtgewichtig")
- Sehr umfangreiche Möglichkeiten (trees, tables, tabbed pane)
- JavaBeans
- Widgets: Subklassen von **JComponent**
- Pluggable Look and Feel
- Model View Controller (MVC) Design Pattern
  - Trennung von View und Model
- Nachteile:
  - eventuell längere Ladezeiten
  - eventuell geringere Ausführungszeit als bei Peer-Klasse

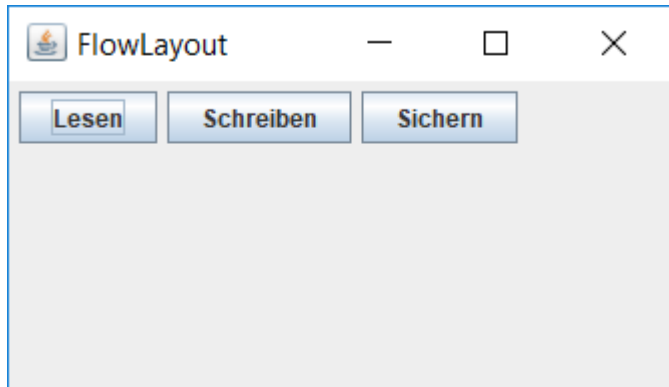


# Layoutmanager

- Ein LayoutManager sorgt für eine plattformunabhängige und dynamische Platzierung der Komponenten im Container
  - FlowLayout
  - BorderLayout
  - GridLayout
  - GridBagLayout
  - ...
- Ohne LayoutManager:
  - `setLayout(null)` : **Nachteil: keine dynamische Anpassung an Fensteränderungen**
  - Lokation und Größe aller Komponenten setzen:
    - `setBounds (int x, int y, int width, int height)`

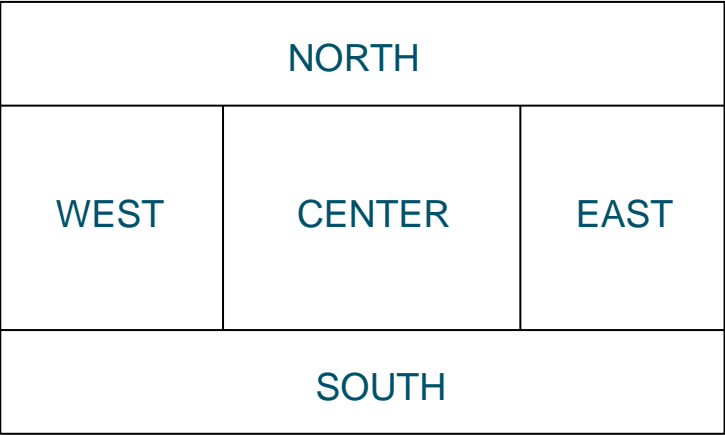
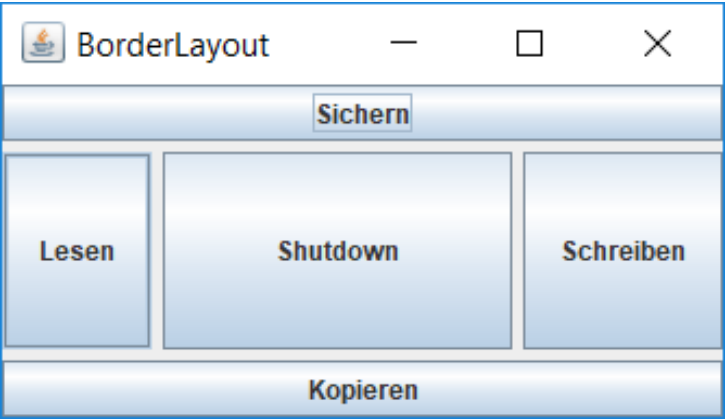


# FlowLayout



- `FlowLayout`:
  - `new FlowLayout()`
  - `new FlowLayout(int hgap, int vgap)`
  - `new FlowLayout(FlowLayout.RIGHT)`
- ordnet Komponenten zeilenweise an.
- in Reihenfolge der `add()`-Befehle
- beliebig viele
- in natürlicher Größe
- eignet sich z. B. für mehrere Buttons nebeneinander

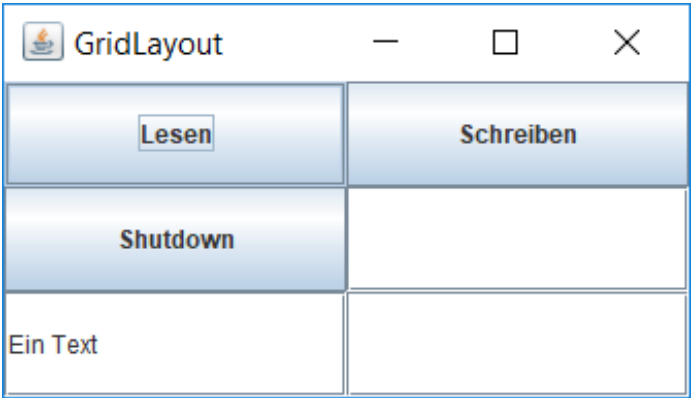
# BorderLayout



- BorderLayout:
  - `new BorderLayout()`
  - `new BorderLayout(int hgap, int vgap)`
- maximal eine Komponente an 4 Rändern und in Mitte
- Zusatzparameter im `add()`-Befehl
  - `add (component, BorderLayout.NORTH)`
  - und analog mit `EAST`, `WEST`, `SOUTH`, `CENTER`
- füllt Container komplett aus
- Komponenten auf ganze Breite bzw. Höhe verzerrt
- eignet sich für die Aufteilung des Fensters in größere Bereiche über oder nebeneinander (Panels, siehe geschachtelte Layouts)



# GridLayout



1	2
3	4
5	6

- GridLayout:
  - `new GridLayout(int rows, int cols);`
  - `new GridLayout(int rows, int cols, int hgap, int vgap);`
- ordnet Komponenten in Gitter an
- in Reihenfolge der `add()`-Befehle
- es wird die genau richtige Anzahl `rows*cols` erwartet
- alle Komponenten gleich groß
- eignet sich z.B. für mehrere Eingabefelder übereinander





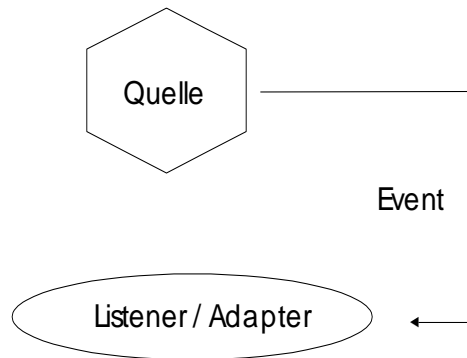
# GridBagLayout



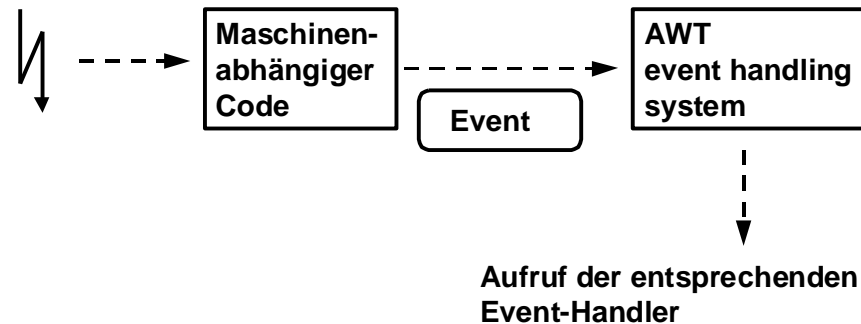
- GridBagLayout :
  - `new GridBagLayout()` ;
- Ausrichtung der Komponenten vertikal und horizontal mit unterschiedlicher Größe
- Jede Komponenten kann eine oder mehrere Zellen des Gitters einnehmen
- Jede Komponenten wird von einer Instanz von GridBagConstraints gesteuert
- Das Constraints-Objekt spezifiziert die genaue Position
- Es wird die natürliche Größe einer Komponente verwendet



# Eventhandling



- Ereignisverarbeitung als Basis der GUIs
- Ein Ereignis (Event) wird von der Quelle (Komponente) zum Listener-Objekt delegiert.
- Der Listener verarbeitet dann das Ereignis
- Wenn der Benutzer ein AWT-Objekt benutzt, erzeugt der plattformabhängige Teil des AWT ein `Event`



- Methoden im Event-Objekt:
  - `getSource()`, `getActionCommand()`, `getX()`, `getY()`, `getClickCount()`, `getKey()`, ...



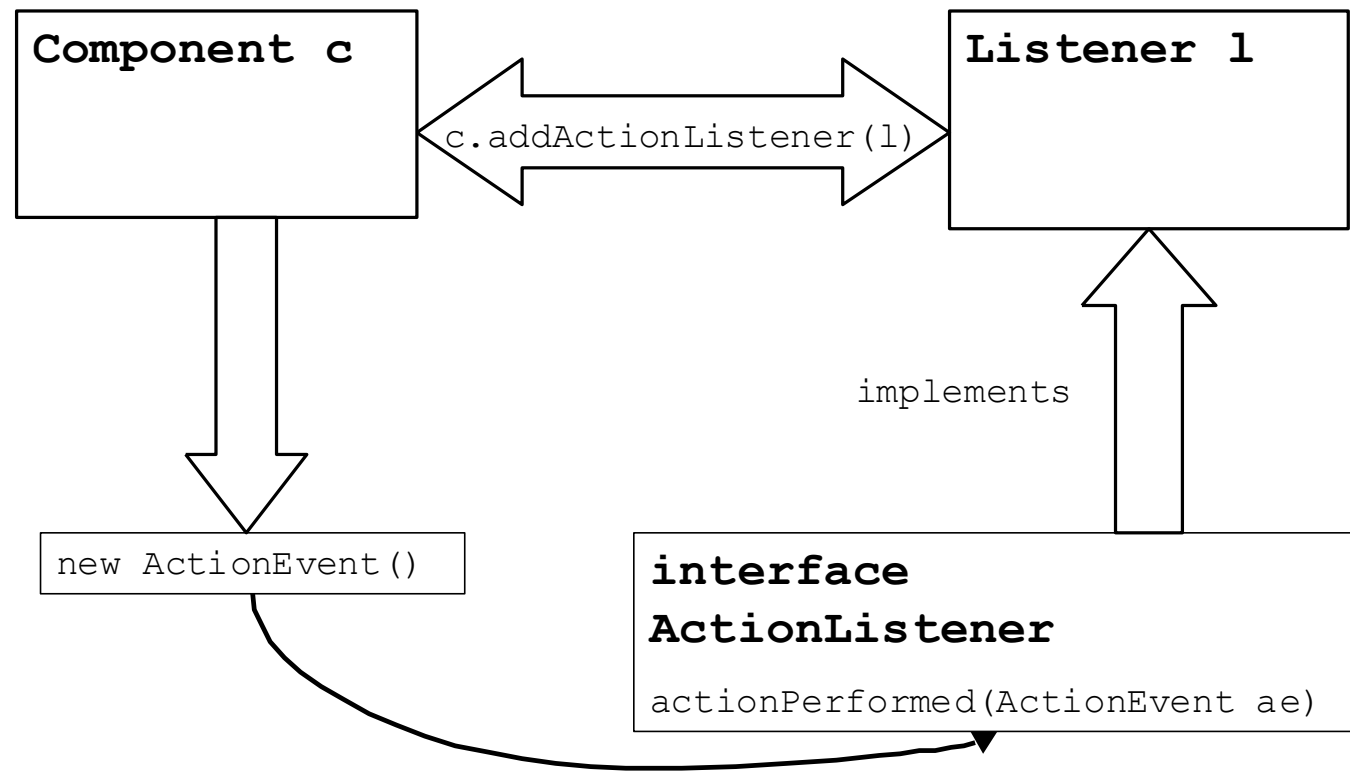
# Event-Klassen, Listener-Interfaces und Methoden

- `ActionEvent`
  - **`ActionListener`**
    - `actionPerformed()`
- `WindowEvent`
  - **`WindowListener`**
    - `windowOpened()`, `windowClosing()`, `windowClosed()`,  
`windowActivated()`, `windowDeactivated()`,  
`windowIconified()`, `windowDeiconified()`
- `MouseEvent`
  - **`MouseListener` und `MouseMotionListener`**
    - `mouseEntered()`, `mouseExited()`,  
`mousePressed()`, `mouseReleased()`, `mouseClicked()`,  
`mouseMoved()`, `mouseDragged()`
- `KeyEvent`
  - **`KeyListener`**
    - `keyPressed()`, `keyReleased()`, `keyTyped()`



# Topologie der Listener

- Ein Listener muss bei einer Komponente registriert werden
- Das Event-Objekt wird von der Komponente zum registrierten Listener gesendet





## Designalternativen für Listener

- Fenster-Komponente (View) und Listener (Controller):
  - zwei getrennte Klassen (View und Controller)
  - eine gemeinsame Klasse (Containerklasse)
  - Listener als innere Klasse
  - Listener als lokale Klasse
  - Listener als anonyme innere Klasse
  - Listener als Lambda-Ausdruck
  - Verwendung einer Adapter-Klasse



# Zusammenfassende Wertung

- Externe Klasse
  - Vorteil: Die Listener können wieder verwendet werden. Mehrere Komponenten können sich am Listener registrieren
  - Nachteil: Der Zugriff auf die Komponenten, die im selben Container liegen, ist schwierig
- Container
  - Vorteil: Einfach zu realisieren, keine weitere Klasse nötig, Zugriff auf alle Attribute und Methoden des Containers
  - Nachteil: schlechtes Design, Delegation für mehrere Komponenten nur mit if-Abfragen
- Innere Klasse
  - Vorteil: klares Design, Zugriff auf Attribute und Methoden des Containers
  - Nachteil: aufwändig
- Anonyme Klasse
  - Vorteil: Listener ist direkt zugeordnet, Delegation, Objektmodell bleibt übersichtlich
  - Nachteil: Nicht wieder verwendbar (ist manchmal sogar gewollt)
- Lambda-Ausdruck
  - Vorteil: Listener ist direkt zugeordnet, nur Inhalt der Methode muss programmiert werden.
  - Nachteil: Nicht wieder verwendbar
- Adapter
  - Vorteil: Nur die benötigten Methoden des Adapters werden überschrieben
  - Nachteil: Geht nicht bei Einfachvererbung



# Kapitel 06: Multithreading



Technische Einführung



Thread definieren



Wechsel zwischen Threads



Lebenszyklus eines Threads



Threadprioritäten



User- und Daemon-Threads



Stoppen eines Threads

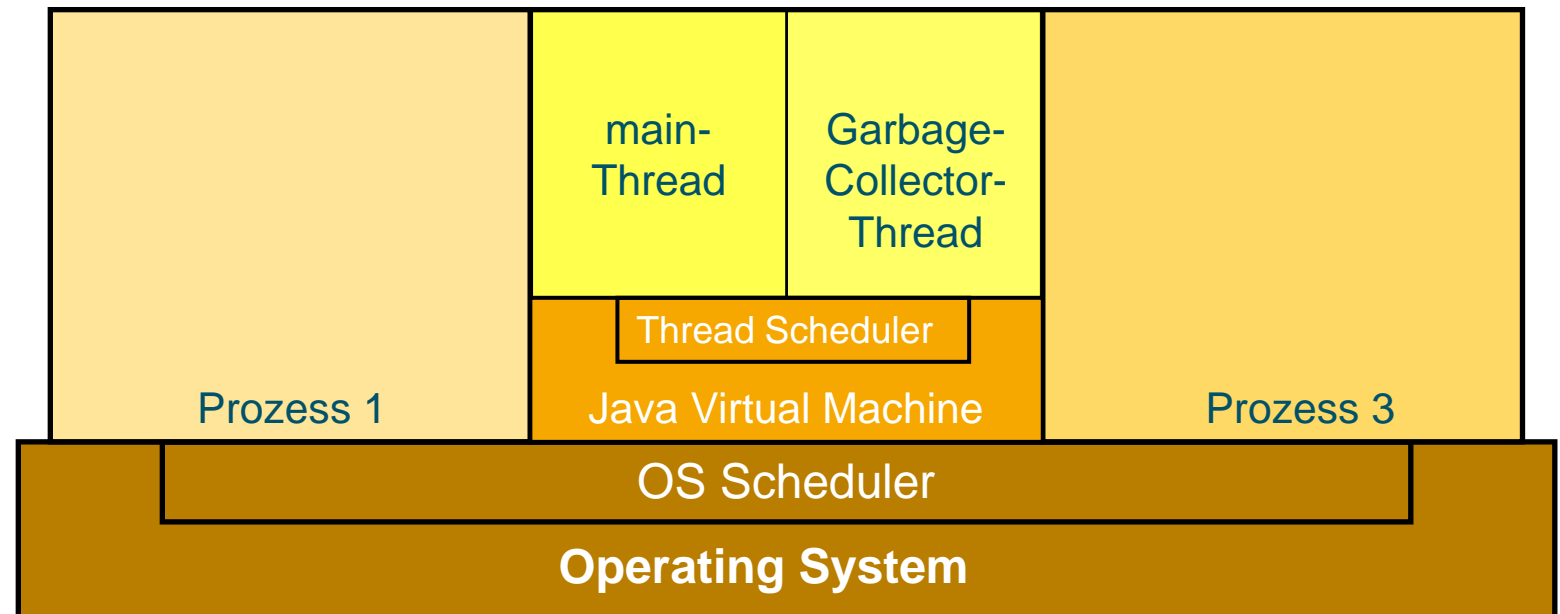


Die Klassen Timer und TimerTask



# Technische Einführung

- Nebenläufige Ausführung von Programmanweisungen
  - analog zu OS-Prozessen
- Gemeinsamer Speicherbereich
  - Threadwechsel ist schneller als Prozesswechsel

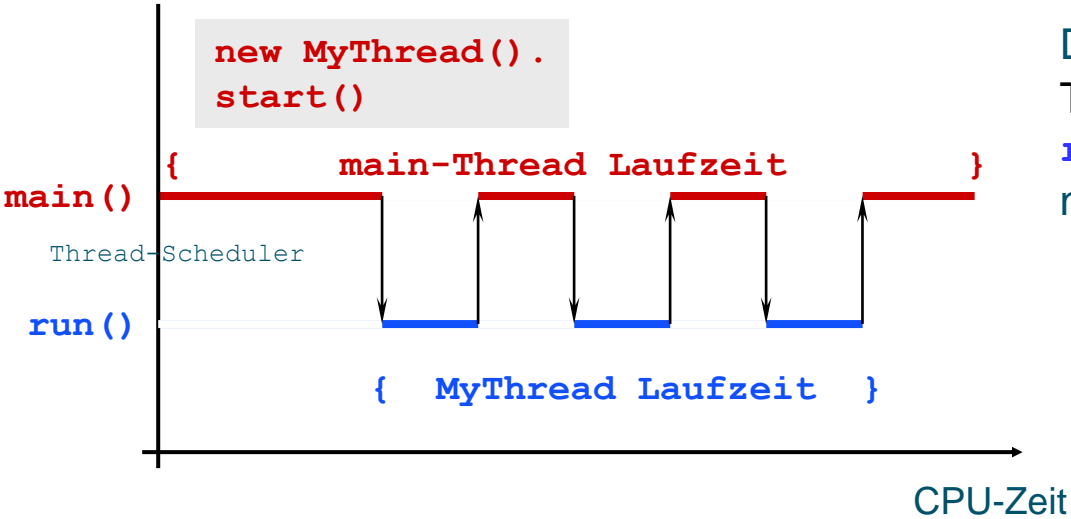






# Technische Einführung

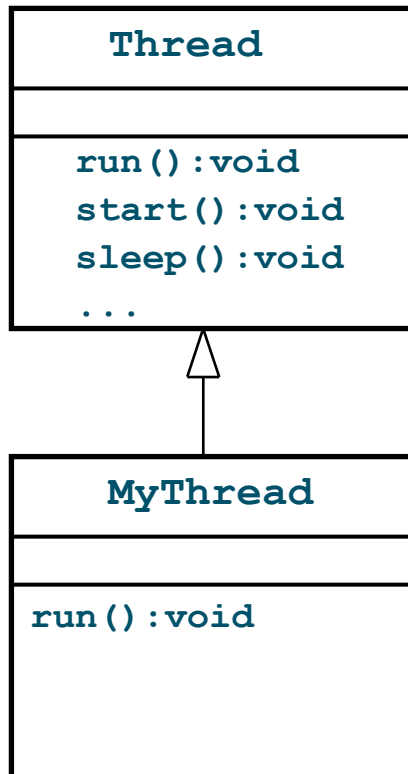
- Ein Thread ist eine Instanz der Klasse `java.lang.Thread` mit einer `run()`-Methode
- Entweder als Subklasse von Thread oder
- als Implementierung des Interfaces Runnable



Die `start()`-Methode von Thread bewirkt, dass die `run()`-Methode als ein neuer Thread läuft.



## Eigenen Thread definieren: Thread vererben



- Programm ist Subklasse von Thread

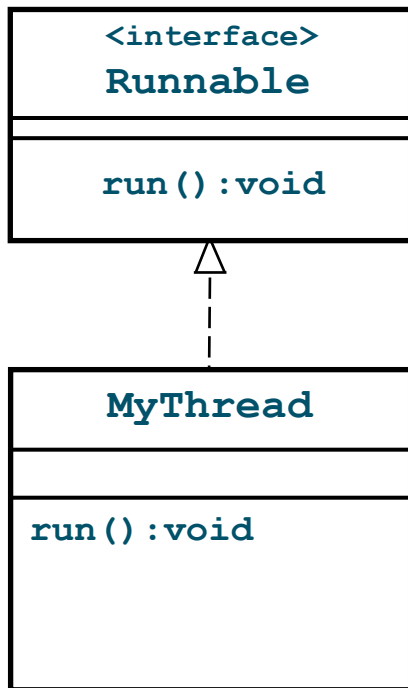
```
public class MyThread extends Thread {  
    public void run() {  
        // do something  
    }  
}
```

- Starten:

```
MyThread prog = new MyThread();  
prog.start();
```



## Eigenen Thread definieren: Runnable implementieren



- Programm implementiert Runnable

```
public class MyThread implements Runnable {
    public void run() {
        // do something
    }
}
```

- Starten:

```
Thread t = new Thread( new MyThread( ) );
t.start();
```

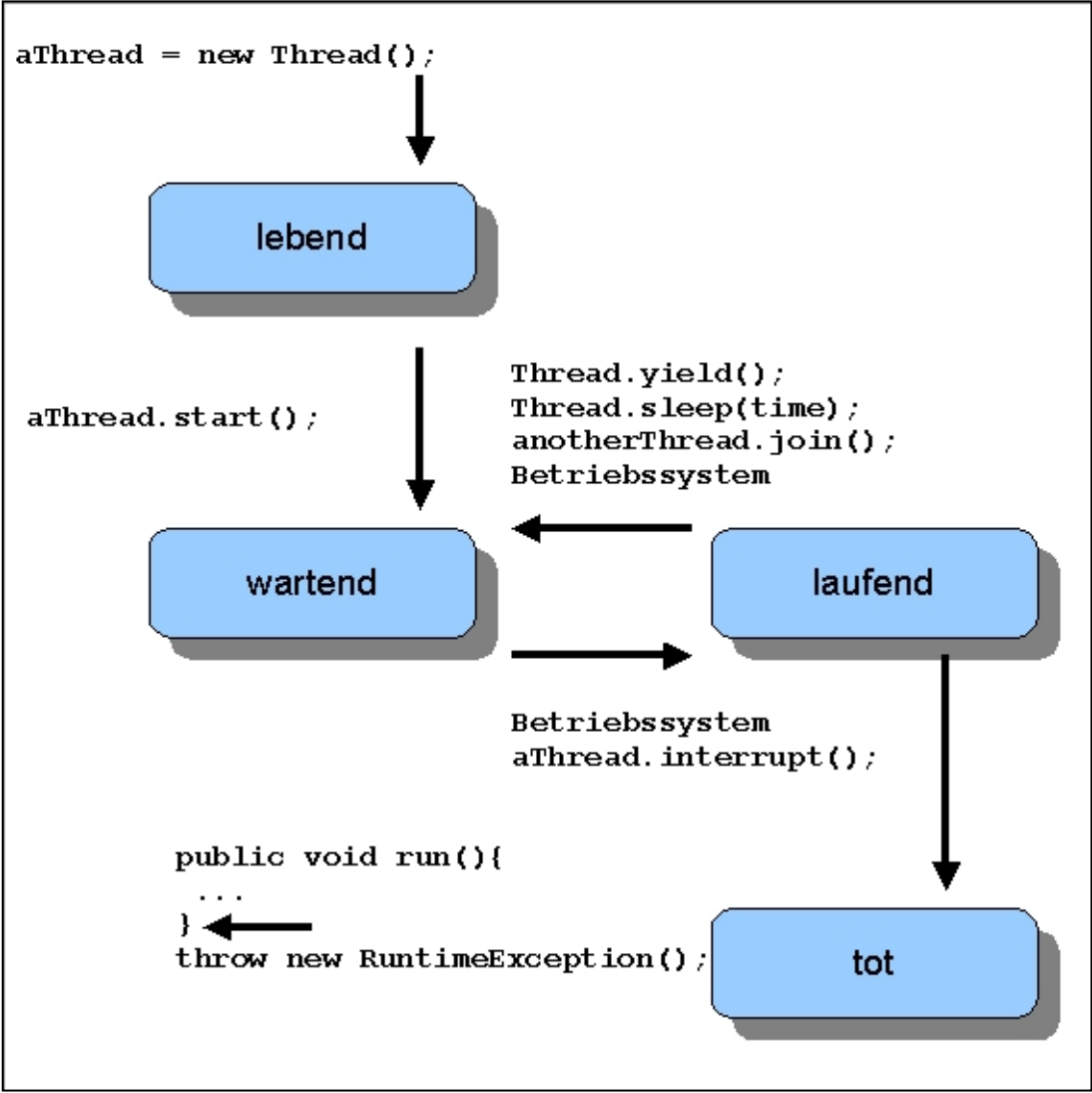


## Wechsel zwischen Threads

- Ein Thread kann sich selbst unterbrechen
  - Damit auch die anderen Threads "drankommen" (co-operative multithreading, non-preemptive)
- statische Methoden
  - `Thread.yield();` // Pause, damit andere Threads laufen können
  - `Thread.sleep(int milliseconds);` // Pause für eine definierte Zeit
  - `Thread.sleep(int milliseconds, int nanoSeconds);`
- Unterbrechung einer definierten Schlaf-Zeit ist möglich
  - `void interrupt()`
  - `static boolean interrupted()`
  - `boolean isInterrupted()`
- Überprüfung, ob der Thread noch lebt
  - `boolean isAlive()`



# Lebenszyklus eines Threads





# Threadprioritäten

- Die Häufigkeit und Länge, mit der ein Thread im laufenden Zustand verweilt, ist abhängig von seiner Priorität
  - `setPriority(int pPriority)`
  - `int getPriority`
  - `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY`, `Thread.MAX_PRIORITY`
- Ein Thread wird erst dann ausgeführt, wenn alle Threads mit höherer Priorität momentan nicht rechenwillig sind.
- Ablauf-Synchronisation durch Prioritäten ist nicht sinnvoll.
- Bevorzugung bestimmter Operationen kann man durch Variieren der Prioritäten der beteiligten Threads schaffen.
  - Ausschlaggebend ist hierbei nicht die absolute Höhe, sondern die Abstufung zwischen den beteiligten Threads.



# User- und Daemon-Threads

- Es gibt User- und Deamon-Threads
  - Der `main`-Thread ist ein User-Thread
  - Vom `main`-Thread gestarteten Threads laufen defaultmäßig als User-Thread
  - Der Garbage-Collector läuft als Deamon-Thread (Hintergrund-Thread)
- Thread-Dämonen sind
  - Threads mit geringer Priorität, die die virtuelle Maschine nicht am „Leben“ erhalten
    - Falls beim Ende des main-Threads nur noch 'Daemons' existieren, wird die VM (mit allen Threads) beendet.
    - Falls noch andere sog. Anwendungs- bzw. User-Threads (`isDaemon == false`) existieren, werden diese nicht beendet.
    - (Die JVM wird beendet, wenn kein User-Thread mehr läuft.)
  - `setDaemon(boolean)` und `boolean isDaemon()`
  - Setzen vor dem Start des Threads



# Stoppen eines Threads

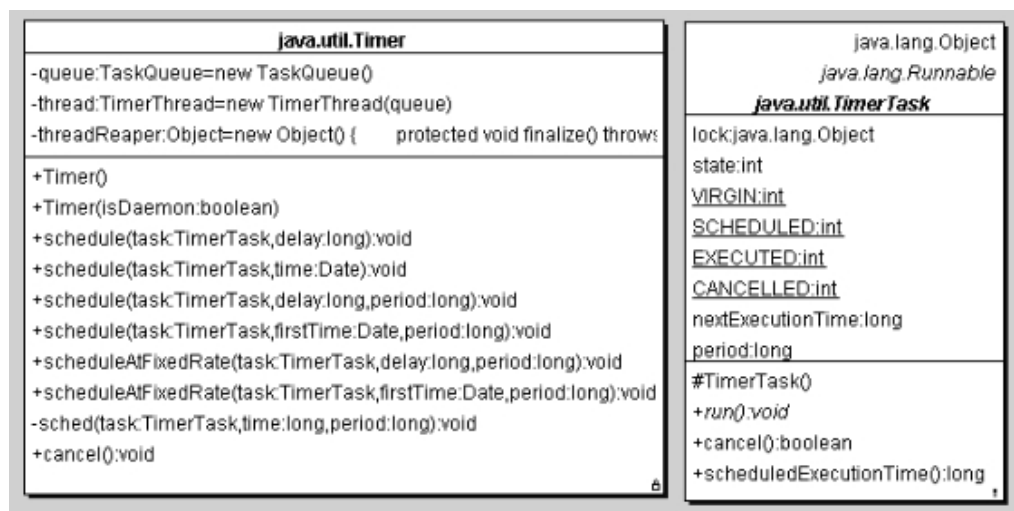
- Nur möglich durch
  - Normale Terminierung der `run()`-Methode
  - werfen einer `java.lang.RuntimeException`
- Stoppen durch Kontroll-Variable innerhalb der `run()`-Methode
  - Dies muss selber programmiert werden!
  - Setzen von „Außen“ durch setter-Methoden vorsehen!
  - Die sofortige Terminierung mit Hilfe der `stop()`-Methode darf **nicht** verwendet werden!





## Die Klassen Timer und TimerTask

- Automatisiertes periodisches Starten eines Prozesses durch `java.util.Timer`
- Definition des Prozesses in einer Subklasse von `java.util.TimerTask`
  - Erfordert Implementierung der `run()`-Methode
- `Timer` ist dazu ausgelegt, sehr viele `TimerTask`-Objekte robust verwalten zu können





## Beispiel mit Timer und TimerTask

```
import java.util.*;

/**
 * Starterklasse für Timer/TimerTask
 */
public class TimerDemo {
    public static void main(String[] args) {

        Timer lTimer = new Timer();

        TimerTask lTask = new TimerTask() {
            public void run() {
                System.out.println("Message from Timer at "
                    + (new Date()).toString());
            }
        };
        lTimer.schedule(lTask, 0, 5000);
    }
}
```



# Kapitel 07: Java und Datenbanken



Überblick



Zwei- und Drei-Schicht Modell



JDBC-API



Treibertypen



Treiber und Datenbank



Verbindung zur Datenbank



JDBC Statements



SQL-Befehle



Auswertung der Ergebnisse



Datenbankzugriff Schritt für Schritt

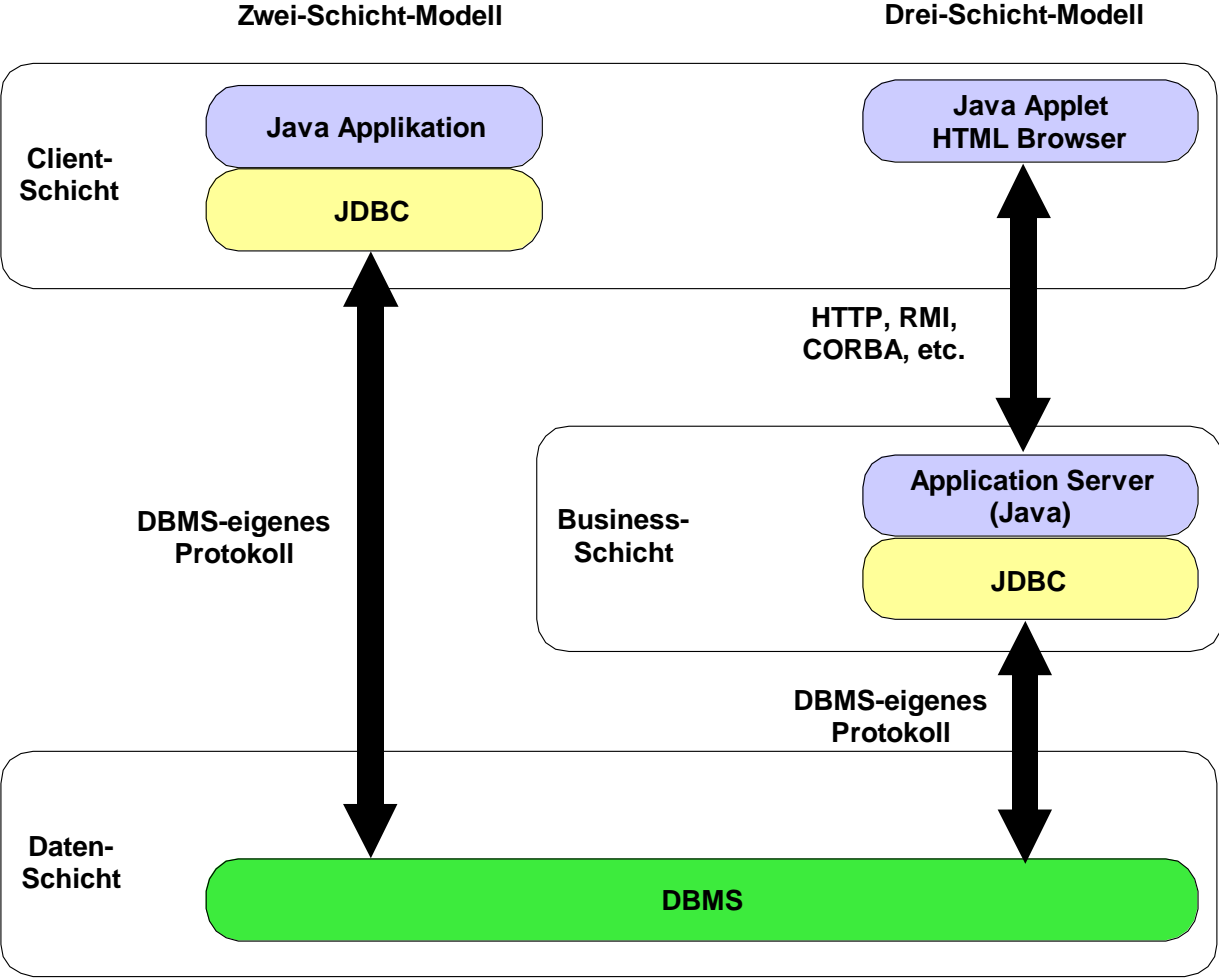


# Überblick

- Strategische Vorteile von JDBC
  - Der Datenbank-Client ist unabhängig von der Zielplattform
  - Die Anbindung an die Datenbank ist unabhängig von der Server-Plattform, da JDBC ein plattformübergreifender Standard ist
  - Der Datenbankclient ist unabhängig vom eingesetzten Datenbankmanagementsystem, da der JDBC-Treibermanager generisch ist
  - Web-Datenbanken werden zunehmend mit HTML-Formularen und JDBC-Aufrufen aus Servlets realisiert



# Zwei- und Drei-Schicht Modell

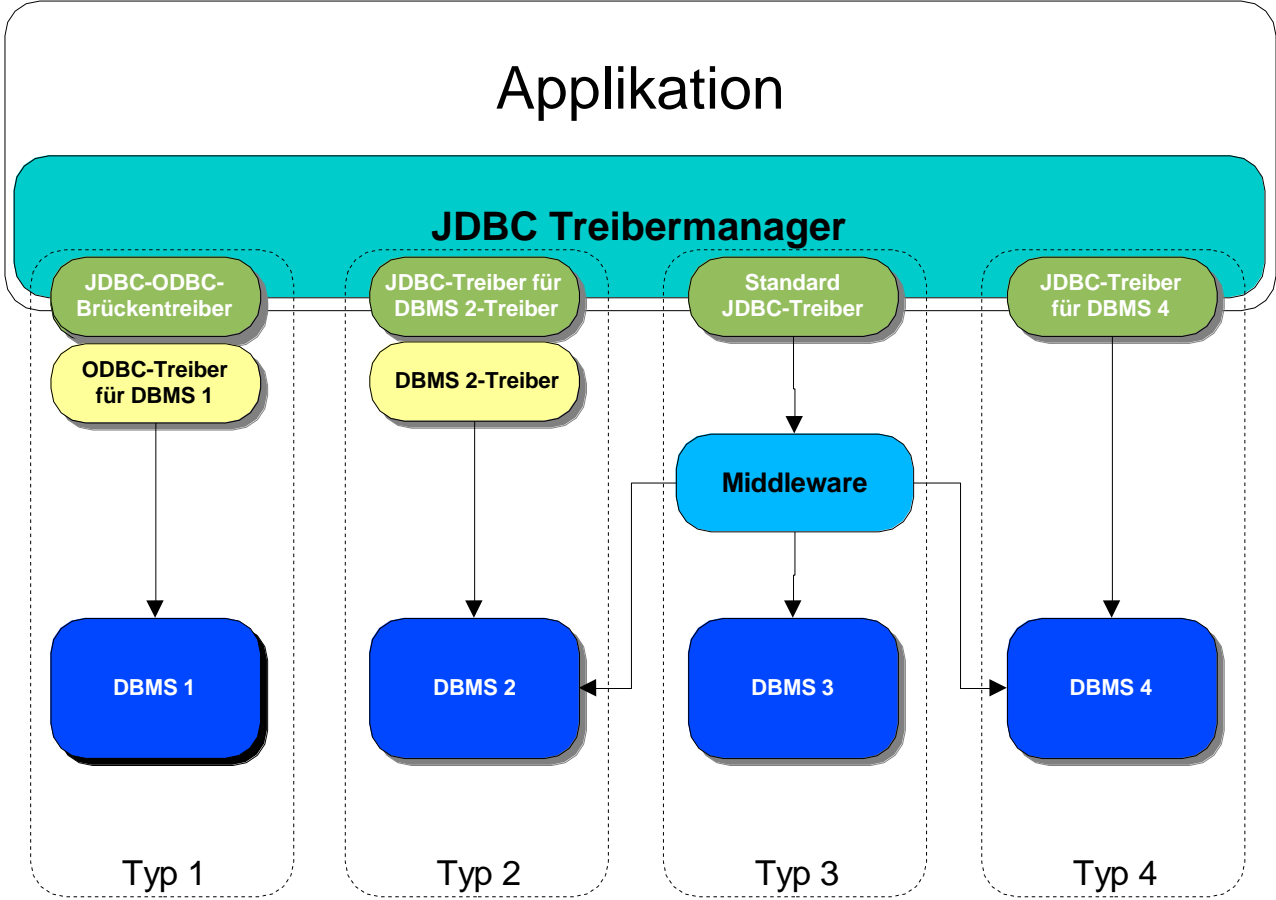




# JDBC-API

- Das JDBC API ist eine Sammlung von Klassen und Interfaces
  - Sie definiert also die Art und Weise, wie der Datenbankzugriff vom Java-Programm aus auszusehen hat
  - Die eigentliche Implementierung muss jemand anderer liefern und zwar jemand, der Wissen über das DBMS hat
- Die Implementierung der JDBC-Schnittstellen nennen wir Treiber
  - Um mit Java auf eine Datenbank zugreifen zu können, brauchen wir einen JDBC-Treiber
  - Ein JDBC-Treiber ist natürlich nichts anderes als eine Klasse
  - Der JDBC-Treiber implementiert die im API definierten Interfaces
  - Der JDBC-Treiber wird meist vom Hersteller des DBMS geliefert.

# Treibertypen





# Treibertypen

- Typ 1:
  - JDBC-Brücke zur anderen DB-API (z. B. ODBC)
  - Benötigt Native-Code auf dem Client.
- Typ 2:
  - JDBC-Treiber ruft die native C/C++ DB-API direkt auf.
  - Benötigt Native-Code auf dem Client.
- Typ 3:
  - JDBC-Treiber sendet Kommandos in einer DB-unabhängigen Sprache über das Netz zur Middleware.
  - Middleware ruft entsprechende API-Kommandos auf.
  - Sehr flexibel, benötigt keinen Native-Code auf dem Client.
- Typ 4:
  - JDBC-Treiber kommuniziert direkt mit der Datenbank (über das Netz).
  - Reine Java-Lösung.





## Typ 1: JDBC-ODBC- Brückentreiber

- Alle JDBC-Befehle werden in ODBC-Befehle übersetzt. Diese werden dann an eine ODBC-fähige Datenbank geschickt
- Vorteile
  - Alle ODBC-Datenquellen sind damit verwendbar
  - Wird kostenlos mit Java SE (bis JDK 7) mitgeliefert
- Nachteile
  - Nicht Internetfähig
  - Langsam (JDBC - ODBC - DBMS API - DB)
  - Installation auf dem Client notwendig
  - Keine reine Java-Lösung
- Beispiel
  - `sun.jdbc.odbc.JdbcOdbcDriver`



## Typ 2: JDBC-DBMS API- Treiber

- Es gibt eine DBMS API in einer anderen Programmiersprache, z.B. C/C++. Alle JDBC-Befehle werden in die Befehle der DBMS API übersetzt, die dann an die Datenbank geschickt werden
- Vorteile
  - Direkter Zugriff auf DBMS API ohne Umweg über ODBC.
  - Schnell
- Nachteile
  - Nicht Internetfähig
  - Installation auf dem Client notwendig
  - Keine reine Java-Lösung
- Beispiel
  - Oracle Thick Driver



## Typ 3: JDBC-Middleware- Treiber

- Der Treiber schickt die JDBC-Befehle nicht direkt an die Datenbank, sondern an eine Middleware. Diese verarbeitet die Befehle und leitet sie an die Datenbank weiter.
- Vorteile
  - Flexibel, da nicht auf ein DBMS spezialisiert. DB-Zugriff wird von der Middleware übernommen.
  - Verwendung in Drei-Schicht-Modellen
  - Internetfähig
  - Reine Java-Lösung
  - Keine Installation auf dem Client notwendig
- Nachteile
  - Langsamer als direkter DB-Zugriff
- Beispiel
  - `COM.cloudscape.core.RmiJdbcDriver`



## Typ 4: JDBC-DB-Treiber in 100% Java

- Funktionsweise: Der DB-Treiber ist in Java geschrieben und erfüllt die JDBC-Schnittstellen.
- Vorteile
  - Verwendung in Zwei-Schicht-Modellen
  - Internetfähig
  - Reine Java-Lösung
  - Keine Installation auf dem Client notwendig
  - Schnellste Variante
- Beispiel
  - Oracle Thin Driver



# Treiber und Datenbank

- JDBC-Treiber sind Java-Klassen, die häufig nur geladen, nicht instanziiert werden müssen
- In einem `static`-Block erfolgt die Anmeldung beim `java.sql.DriverManager`
- Alternativ: Umgebungsvariable `jdbc.drivers`
  - Durch Doppelpunkte getrennte Liste von voll qualifizierten Klassennamen

Für die Verwendung einer Datenbank wird benötigt:

- SQL-Bibliotheken
  - `import java.sql.*;`
- URL mit Protokoll, Host, Portnummer, Datenbankname
  - `String url = "jdbc:subprotokoll://host:port/DBName";`
- Treiber laden
  - `Class.forName("voll qualifizierter Klassenname des Treibers");`



# Verbindung zur Datenbank

- `java.sql.Connection` = eigene Verbindung zur Datenbank
- Ein `Connection`-Objekt wird vom `DriverManager` erzeugt
  - `Connection con = DriverManager.getConnection(url, „user“, „pwd“);`
- Eine `Connection` wird benutzt:
  - Um SQL-Statements zu erzeugen und verwalten
    - `Statement s = con.createStatement();`
  - Um den Zugriff zu steuern
    - `con.setReadOnly(true);`
    - `con.setAutoCommit(false);`
    - ...
  - Um Transaktionen zu steuern
    - `con.commit();`
    - `con.rollback();`



# JDBC Statements

- Statements werden innerhalb einer Connection erzeugt.
- `java.sql.Statement`
  - SQL wird an die DB gesandt und dort wird für jedes execute ein Bearbeitungsplan erstellt
  - Relativ langsam, aber flexibel
- `java.sql. PreparedStatement`
  - SQL wird an die DB gesandt und dort wird nur einmal ein Bearbeitungsplan erstellt
  - Schneller, besonders für oft wiederholte Abfragen
- `java.sql.CallableStatement`
  - Vorhandene SQL-Programme (stored procedures) werden aufgerufen
  - Am schnellsten, SQL-Code muss auf dem Server verfügbar sein



# SQL-Befehle

- verschiedene Methoden je nach Rückgabewert
- in Statement:
  - `ResultSet executeQuery (String sql)`
  - `int executeUpdate (String sql)`
  - `boolean execute (String sql) + getResultSe() / getUpdateCount()`
- in PreparedStatement und CallableStatement:
  - SQL-String beim Anlegen des Statements + set-Methoden
  - `ResultSet executeQuery ()`
  - `int executeUpdate ()`
  - `boolean execute () + getResultSe() / getUpdateCount()`





# Auswertung der Ergebnisse

beforeFirst

id	Pers_NR	Nachname	Vorname
1	K23408	Müll	Gerd
2	L77634	Abfall	Peter

- `ResultSet` = Ergebnis einer DB-Abfrage (Query)
- abhängig vom der JDBC-Version wird ein Cursor zur Navigation in der Ergebnismenge verwaltet
- Initial steht der Cursor vor der ersten Zeile der Ergebnismenge
- Methoden für Zugriff auf Daten

```
public boolean next()           //Existiert noch eine gültige
                                Reihe?
Typ getTyp(String spalte)      //Liefert einen Typ zurück.
Typ getTyp(int spalte)         //Liefert einen Typ zurück.
first()
last()
```

- Beispiel :  

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Tabelle");
while(rs.next()) {
    String sNachname = rs.getString("Nachname");
    String sVorname = rs.getString("Vorname");
}
```



# Datenbankzugriff

## Schritt für Schritt

### 1. Treiber laden

```
Class.forName( ... )
```

### 2. URL definieren

z.B. "jdbc:odbc:Tabelle" oder

### 3. Verbindung zur Datenbank herstellen

```
con = DriverManager.getConnection(url, user, password);
```

### 4. Statement besorgen

```
stmt = con.createStatement();
```

### 5. Anfrage in SQL

```
rs = stmt.executeQuery("SELECT * FROM Tabelle");
```

### 6. Ergebnisse ausgeben

```
while(rs.next()) {  
    int i = rs.getInt(1);  
    String s = rs.getString("Vorname");  
}
```



# Kapitel 08: Anhang



Eclipse



Eclipse: Projekterzeugung




Dokumentation




Javadoc

# Eclipse


■ [www.eclipse.org](http://www.eclipse.org)




Get **Eclipse IDE** 

Install your favorite desktop IDE packages.

[Download 64 bit](#)




MembersWorking GroupsProjectsMore▼


[Download](#)

## The Platform for Open Innovation and Collaboration


The Eclipse Foundation provides our global community of individuals and organizations with a mature, scalable and commercially-friendly environment for open source software collaboration and innovation.




Members



Working Groups



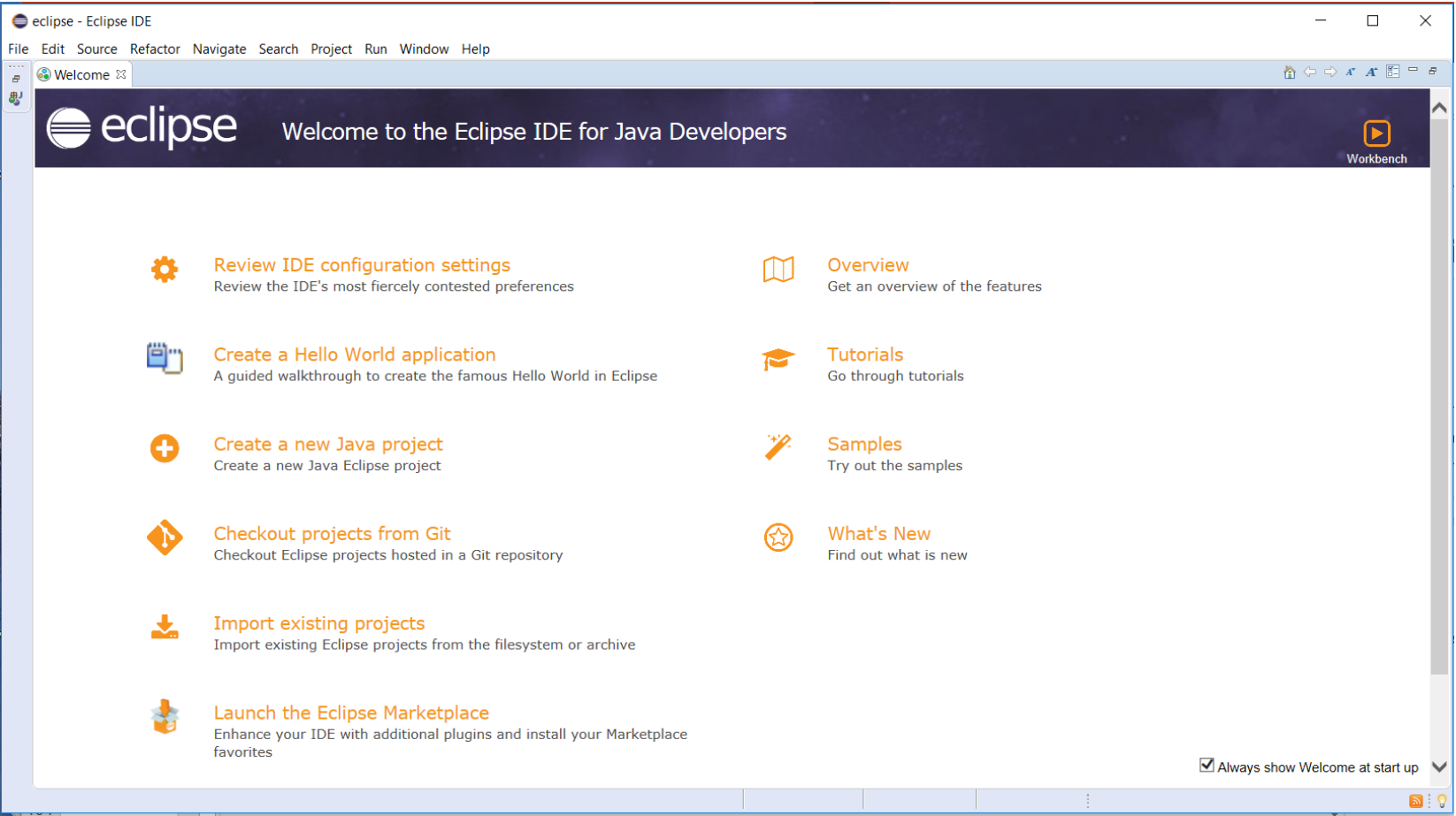
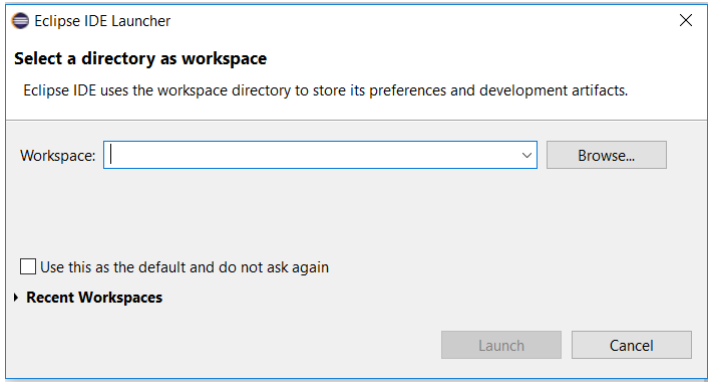
Discover Projects



Business Value

### We Want to Hear from You!

# Eclipse



# Eclipse

New Java Project

Create a Java Project

Enter a project name.

Project name:

☒ Use default location

Location: C:\temp\eclipse

Browse...

JRE

☒ Use an execution environment JRE:

JavaSE-1.8

☐ Use a project specific JRE:

jre1.8.0\_192

☐ Use default JRE (currently 'jre1.8.0\_192')

[Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files

[Configure default...](#)

Working sets

☐ Add project to working sets

New...

Working sets:

Select...

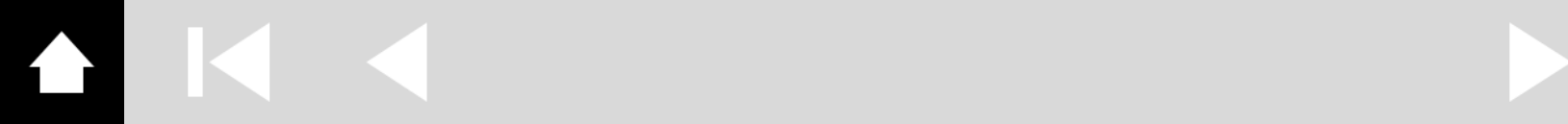
?

< Back

Next >

Finish

Cancel



# Online- Dokumentation

- <http://docs.oracle.com/javase/8/docs/api/index.html>
- <http://docs.oracle.com/javase/9/docs/api/index.html>
- <http://docs.oracle.com/javase/10/docs/api/index.html>

OVERVIEW

MODULE

PACKAGE

CLASS

USE

TREE

DEPRECATED

INDEX

HELP

PREV

NEXT

FRAMES

NO FRAMES

ALL CLASSES

Java® Platform, Standard Edition & Java Development Kit

Version 10 API Specification

This document is divided into three sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the J names start with `jdk`.

JavaFX

The JavaFX APIs define a set of user-interface controls, graphics, media, and web packages for developing rich client applications.

All Modules

Java SE

JDK

JavaFX

Other Modules

Module	Description
<code>java.activation</code>	Defines the JavaBeans Activation Framework (JAF) API.
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
<code>java.corba</code>	Defines the Java binding of the OMG CORBA APIs, and the RMI-IIOP API.
<code>java.datatransfer</code>	Defines the API for transferring data between and within applications.



# Javadoc

- Eigene Dokumentationen können automatisch generiert werden durch das im JDK enthaltene Tool javadoc.
- javadoc durchsucht Quelldateien nach Klassen, Methoden und Kommentaren.
- javadoc-Kommentare werden mit dem speziellen Kommentar `/** ... */` eingegrenzt.
- Ein Kommentar besteht aus beliebigem Text im HTML-Format.
- Zusätzlich erkennt javadoc noch spezielle Tags:
  - `@see Klasse`
  - `@see Klasse#Methode`
  - `@version Versionsname oder -nummer`
  - `@author Name des Autors`
  - `@return Beschreibung des Rückgabewertes`
  - `@throws Exceptionbeschreibung`
  - `@param Parameterbeschreibung`



# Copyright und Impressum

© Integrata AG

Integrata AG  
Zettachring 4  
70567 Stuttgart

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.**